IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

BEFORE THE PATENT EXAMINING OPERATION

ATTN'Y DOCKET NO.:	ETS-TCA
APPLICATION OF:	PETER BRITTINGHAM, MARY E. MORLEY, MARK K. SINGLEY, MARK G. ZELMAN, KRISHNA N. JHA, JAMES H. FIFE, ROBERT L. RARICH, IRVIN R. KATZ, RANDY E. BENNETT
FOR:	COMPUTER-BASED TEST-ITEM GENERATION AND CLONING

PROLOG SOURCE CODE APPENDIX (PROLOG SCA 1-95)

EI469815916US

PROLOG SOURCE CODE APPENDIX
TABLE OF CONTENTS[1]

```
' HLP4lib.p4
%%
%%	HLP4lib.p4: library of PrologIV accessory relations useful in High-Level API
%%
%%	We follow the convention that the [conventional] result is bound to the first argument,
%%		so that these relations can be easily called as functions.
%%
%%	Note: BEWARE: Interval operators (e.g. ./., .*., ...) sometimes give rise to errors (e.g.
divide-by-zero error)
%%		in VB when used in conjunction with VB.
%%	Note: BEWARE: intsplit/realsplit on unbounded variabes sometimes give rise to errors
(e.g. overflow error)
%%		in VB when used in conjunction with VB.

%% HLAPI-library functions/relations

		%% debug_print(-Result, +List):	Result is true if it prints all the elements of
the list, ended by nl
debug_print(true, [])	:-	nl.
debug_print(Result, [A| Rest])		:-
		write(A), write(','),
		debug_print(Result, Rest).

		%% var_with_precision(+V, +Prec): Succeeds if var V is an integer multiple of
given Precision.
		%%	(Do NOT introduce intsplit(N) here - gives rise to overflow error in VB.)
var_with_precision(V, Prec)	:-
	int(N),
	V = N*Prec.

		%% even(-Result, +N): Result is true if N is an even integer.
even(true, N)		:-
	even(N).

		%% odd(-Result, +N): Result is true if N is an odd integer.
odd(true, N)		:-
	odd(N).

		%% max(-Result, +A, +B): Result is maximum of A and B. -- already provided in
PrologIV
		%% max(-Result, [A, B, C, ...]): Result is maximum of array of numbers [A, B, C,
...].
max(Result, [A| Rest])		:-
		max_array_aux(Result, A, Rest).
max_array_aux(A, A, []).
```

```
max_array_aux(Result, A, [B|Rest]) :-
        max(Rmax, A, B),
        max_array_aux(Result, Rmax, Rest).

        %% min(-Result, +A, +B): Result is minimum of A and B. -- already provided in PrologIV
        %% min(-Result, [A, B, C, ...]): Result is minimum of array of numbers [A, B, C, ...].
min(Result, [A| Rest])          :-
        min_array_aux(Result, A, Rest).
min_array_aux(A, A, []).
min_array_aux(Result, A, [B|Rest]) :-
        min(Rmin, A, B),
        min_array_aux(Result, Rmin, Rest).

        %% mean(-Mean, A, B): Mean is mean of numbers A & B
mean((A+B)/2, A, B).

        %% mean(-Mean, [A, B, C, ...]): Mean is mean of array of numbers [A,B,C,...]
mean(A, [A]).
mean(Sum/Size, [A|Rest])        :-
        array_sum(Sum, [A|Rest]),
        size(Size, [A|Rest]).

        %% median(-Med, +[A, B, C, ...]): Med is the median of array of numbers [A, B, C, ...]
median(Med, [A|Rest])           :-
        sort(SortedList, [A|Rest]),
        size(Size, SortedList),
        pick_midlist(Med, SortedList, Size).
pick_midlist(Mid, List, ListSize)       :-
        odd(ListSize),
        index(Mid, List, ((ListSize-1)/2)+ 1).
pick_midlist((Mid1+Mid2)/2, List, ListSize)       :-
        even(ListSize),
        index(Mid1, List, ListSize/2),
        index(Mid2, List, (ListSize/2)+1).

        %% gcd(-GCD, +A, +B):     GCD is gcd of A and B. -- until PrologIV provides it.
gcdtemp(A./.Num, A, B)          :-
        int(A), int(B), B gtlin 0,
        numden(A./.B, Num, _Den).

        %% lcm(-LCM, +A, +B):     LCM is lcm of A and B. (lcm(A, B)= A* B/
```

```
gcd(A,B) )-- until PrologIV provides it.
lcmtemp((A.*.B)./.GCD, A, B)        :-
                int(A), int(B), B gtlin 0,
                gcdtemp(GCD, A, B).

                %% mod(-Mod, N, D): Mod= N modulo M.
modtemp(Mod, N, D)                      :-
                int(N), int(D), D gtlin 0,
                modulo(N, D, Mod).

                %% divmod(-DivMod, N, D): True if DivMod= [N Div D, N Mod D]
divmod([Div, Mod], N, D)    :-
                int(N), int(D), D gtlin 0,
                intdiv(Div, N, D),
                modulo(N, D, Mod).

                %% numdentemp([-N, -D], R): True if N/D = R (where N and D are integers)
numdentemp([N, D], R)               :-
                real(R),
                numden(R, N, D).

                %% quotnumden([-Q, -N, -D], R): True if (Q+(N/D)) = R (where Q, N and D are
integers)
quotnumden([Q, N, D], R)    :-
                real(R),
                numden(R, N1, D),
                divmod([Q, N], N1, D).

                %% sqrt(Sqrt, N):      Sqrt is square-root of N -- already provided by PrologIV
                %% is_perfect_square(-Result, +N): succeeds (and sets Result to true) if N is a
perfect square.
is_perfect_square(true, N)              :-
                int(N), int(Sqrt),
                sqrt(Sqrt, N).
                %% isnot_perfect_square(-Result, +N): succeeds (and sets Result to true) if N is
NOT a perfect square.
isnot_perfect_square(true, N)           :-
                int(N), nint(Sqrt),
                sqrt(Sqrt, N).

                %% cubert(-CubeRoot, +N): CubeRoot is cube-root of N
cubert(CubeRt, N)               :-
                int(N),
                root(CubeRt, N, 3).
                %% is_perfect_cube(-Result, +N): succeeds (and sets Result to true) if N is a
```

```
perfect cube.
is_perfect_cube(true, N)            :-
            int(N), int(Cbrt),
            cubert(Cbrt, N).
            %% isnot_perfect_cube(-Result, +N): succeeds (and sets Result to true) if N is
NOT a perfect cube.
isnot_perfect_cube(true, N)         :-
            is_perfect_cube(true, N), !, fail.
isnot_perfect_cube(true, N)   :-    %% this alone does not work well because of roundoff
problems
            int(N), nint(Cbrt),
            cubert(Cbrt, N).

            %% is_prime(-Result, +N): succeeds (and sets Result to true) if N is a
prime-number.
is_prime(true, 2). is_prime(true, 3).  %% base cases (note that 1 is NOT considered prime.)
is_prime(true, N)             :-
            int(N), abs(AbsN, N), AbsN gt 3,
            sqrt(RlSqRoot, AbsN),
            ceil(SqRoot, RlSqRoot),
            aux_check_prime(AbsN, 2, SqRoot).
aux_check_prime(N, CurDivisor, MaxDivisor)     :-
            CurDivisor gt MaxDivisor.
aux_check_prime(N, CurDivisor, MaxDivisor)     :-          %% improve it later
            CurDivisor le MaxDivisor,
            modulo(N, CurDivisor, Mod), Mod gt 0,
            aux_check_prime(N, CurDivisor + 1, MaxDivisor).

            %% isnot_prime(-Result, +N): succeeds (and sets Result to true) if N is NOT a
prime-number.
isnot_prime(true, N)          :-
            nint(N), real(N).
isnot_prime(true, N)          :-
            int(N), abs(AbsN, N), AbsN gt 3,
            sqrt(RlSqRoot, AbsN),
            ceil(SqRoot, RlSqRoot),
            aux_check_nonprime(AbsN, 2, SqRoot).
aux_check_nonprime(N, CurDivisor, MaxDivisor)  :-
            CurDivisor le MaxDivisor,
            modulo(N, CurDivisor, 0), !.
aux_check_nonprime(N, CurDivisor, MaxDivisor)  :-
            CurDivisor le MaxDivisor,
            aux_check_nonprime(N, CurDivisor + 1, MaxDivisor).

            %% nth(-NthElem, +N, +List): NthElem is the Nth element of the given List;
```

```
(N==1 for first elem.)
nth(NthElem, N, List)            :-
        index(NthElem, List, N).


        %% permute(-PermutedList, +List): PermutedList is a permutation of List.
permute(PermutedList, List)  :-
        permute_aux(PermutedList, [], List).

permute_aux(PermutedList, PermutedList, []).
permute_aux(PermutedList, APermutation, RightList)        :-
        an_elem_and_rest(Elem, Rest, RightList),
        permute_aux(PermutedList, [Elem| APermutation], Rest).

        %% factorial(-Factorial, +N): Factorial= N!
        %% naive implementation: factorial(1, 0). factorial(N* Fact, N)      :-
factorial(Fact, N-1).
        %%     we just list them here which also gives a bidirectional relationship.
factorial(1, 0). factorial(1, 1). factorial(2, 2).
factorial(6, 3). factorial(24, 4).        factorial(120, 5).
factorial(720, 6).        factorial(5040, 7).       factorial(40320, 8).
factorial(362880, 9).             factorial(3628800, 10).
factorial(39916800, 11).          factorial(479001600, 12).
factorial(6227020800, 13).        factorial(87178291200, 14).
factorial(1307674368000, 15).             factorial(20922789888000, 16).
factorial(355687428096000, 17).           factorial(6402373705728000, 18).
factorial(121645100408832000, 19). factorial(2432902008176640000, 20).
factorial(N* Fact, N)  :- number(N), N gt 20, factorial(Fact, N-1).

        %% non-naive implementation of factorial - not used
%% factorial(1, 0).
%% factorial(Factorial, N)          :-
%%              int(N), N gt 0,
%%              factorial(Factorial, N, N.-.1).
%%factorial(Factorial, Factorial, 0).
%%factorial(Factorial, FactSoFar, N)          :-
%%              N gt 0,
%%              factorial(Factorial, FactSoFar.*.N, N.-.1).

    %% enumerate(-R, +Min, +Max, +Step): enumerate (any var) R between
(closed-interval) [Min, Max] by Step.
enumerate(R, Min, Max, Step)          :-
        min(RMin, Min, Max),
        max(RMax, Min, Max),
        RMin le RMax,
```

```
                Step gt 0,
                enumerate_aux(R, RMin, RMax, Step).

        %% enumerate_int(-I, +Min, +Max, +Step): enumerate (integer var) I between
(closed-interval) [Min, Max] by Step.
enumerate_int(I, Min, Max, Step)      :-
                int(I),
                min(RMin, Min, Max),
                max(RMax, Min, Max),
                RMin le RMax,
                ceil(IMin, RMin),
                floor(IMax, RMax),
                IMin le IMax,
                floor(IStep, Step),      %% should IStep be floor, or ceiling ??
                IStep gt 0,
                enumerate_aux(I, IMin, IMax, IStep).

        %% enumerate_aux(-R, +Min, +Max, +Step): enumerate (any var) R between
(closed-interval) [Min, Max] by Step.
        %%      Note that we enumerate from both ends i.e. from Min and from Max ends.
        %%      Note that increasing the no. of partitions requires large choice-stack and heap
sizes.
        %%      (We can adjust the various stack & heap sizes - but just that there is a cost to more
partitions.)
enumerate_aux(R, Min, Max, Step)   :-
                StepsCnt= (Max- Min)/Step,
                StepsCnt le 4,                  %% le 4 partitions => le 5 points in the range
                !,                              %% small range - simple enumeration
                enumerate_aux_simple(R, Min, Min, Max, Step).
enumerate_aux(R, Min, Max, Step)   :-     %% large range - interleave the enumeration
                StepsCnt= (Max- Min)/Step,  %% adjust the Max
                floor(NMax, StepsCnt),
                PartitionStepCnt = StepsCnt/6,        %% split the steps-range into 5 partitions
                floor(Inc, PartitionStepCnt),         %% problem with ceil/2 when numbers are
small
                enumerate_aux_interleave(R,           %% interleave the 10 partitions
                        [[Min, Min+ Inc* Step, inc],            %% 1st: enumerate up
                         [Min+(Inc+1)*Step, Min+2*Inc*Step, dec],%% 2nd: enumerate down
                         [Min+(2*Inc+1)*Step,Min+3*Inc*Step,inc],
                         [Min+(3*Inc+1)*Step,Min+4*Inc*Step,dec],
                         [Min+(4*Inc+1)*Step,Min+NMax*Step,dec]],
                        Step).

                %% enumerate R simply between Min & Max by Step
enumerate_aux_simple(R, R, _Min, _Max, _Step).
```

```
enumerate_aux_simple(R, Prev, Min, Max, Step)     :-
                (Prev+ Step) le Max,
                enumerate_aux_simple(R, Prev+Step, Min, Max, Step).

                %% enumerate in an interleaved fashion - from all partitions - upward or
downward
enumerate_aux_interleave(RMin, [[RMin, RMax, inc]| _Rest], _Step):-
                RMin le RMax.
enumerate_aux_interleave(RMax, [[RMin, RMax, dec]| _Rest], _Step):-
                RMin le RMax.

enumerate_aux_interleave(R, [[RMin, RMax, _IncOrDec]| Rest], Step):-
                (RMin+ Step) gt RMax, !,      %% partition done - drop it from the partitions-list
                random_shuffle(Shuffled, Rest),
                enumerate_aux_interleave(R, Shuffled, Step).

enumerate_aux_interleave(R, [[RMin, RMax, inc]| Rest], Step):-
                random_shuffle(Shuffled, Rest),
                ncons(ResLst, [RMin+ Step, RMax, inc], Shuffled),
                enumerate_aux_interleave(R, ResLst, Step).
enumerate_aux_interleave(R, [[RMin, RMax, dec]| Rest], Step):-
                random_shuffle(Shuffled, Rest),
                ncons(ResLst, [RMin, RMax- Step, dec], Shuffled),
                enumerate_aux_interleave(R, ResLst, Step).


                %% select_r_of_n_ordered(-Pn_r, +N, +R): Select no. of permutations of setsize
N selecting R at a time
select_r_of_n_ordered(Nfact./. Rfact, N, R)  :-
                int(N), int(R), N gt 0, R gt 0, N ge R,
                factorial(Nfact, N),
                factorial(Rfact, R).

                %% select_r_of_n(-Pn_r, +N, +R): Select no. of combinations of setsize N
selecting R at a time
select_r_of_n(Perm./. NRfact, N, R)  :-
                select_r_of_n_ordered(Perm, N, R),
                factorial(NRfact, N-R).

                %% abs(-AbsN, +N): AbsN is absolute number, N. -- already provided by
PrologIV


                %% isbound(-Result, +X): Result is true if the given (numerical variable/value) X
is bound on the low and/or upper side.
```

```
isbound(true, X)      :-      isbound(X).
isbound(X) :- glb(X,_) , !.
isbound(X) :- lub(X,_).

                %% isnotbound(-Result, +X): Result is true if the given (numerical variable) X is
NOT bound on either the low or the upper side.
isnotbound(true, X)   :-      isnotbound(X).
isnotbound(X) :- isbound(X), !, fail.
isnotbound(X).

%% auxiliary functions
                %% even(N):  true if N is even
even(2.*.X)     :-      int(X).

                %% odd(N):   true if N is odd
odd(N)          :-      int(N), N= 2.*.X, nint(X).

                %% reverse(-ReverseList, +List): ReverseList is reverse-ordered List
reverse(ReverseList, List)     :-
                reverse(ReverseList, [], List).
reverse(ReverseList, ReverseList, []).
reverse(ReverseList, RevListSoFar, [A|Rest])        :-
                reverse(ReverseList, [A|RevListSoFar], Rest).

                %% ncons(-ResList, +A, +List): true if ResList= List+ [A].
ncons(ResList, A, List)                 :-
                append(ResList, List, [A]).

                %% append(-AppendedList, List1, List2): AppendedList is result of appending
lists List1 & List2.
append(A, [], A).
append([A|Result], [A|Rest], B)        :-
                append(Result, Rest, B).

                %% random_shuffle(-ShuffledList, +List): Succeeds if List is random-shuffled
into SuffledList
random_shuffle(ShuffledList, List)   :-
                random_shuffle(ShuffledList, [], List).

random_shuffle(ShuffledList, ShuffledList, []).
random_shuffle(ShuffledList, ShuffledListSoFar, [A| Rest])          :-
                brandom(Random),
                ((Random=1)  ->
                random_shuffle(ShuffledList, [A|ShuffledListSoFar], Rest);
                (ncons(NewShuffledListSoFar, A, ShuffledListSoFar),
```

```
                random_shuffle(ShuffledList, NewShuffledListSoFar, Rest) )
                ).

                %% array_sum(-ArraySum, +Array): Sum is the sum of the array-elements of
Array
array_sum(0, []).
array_sum(ArraySum, [A|Rest])       :-
                number(A),
                array_sum(ArraySum, A, Rest).
array_sum(ArraySum, ArraySum, []).
array_sum(ArraySum, SumSoFar, [A|Rest]) :-
                number(A),
                array_sum(ArraySum, SumSoFar.+.A, Rest).

                %% sort(-SortedArray, +Array): Array (of numbers) is sorted (in ascending order)
into SortedArray
sort([], []).
sort(SortedArray, [E|Array])  :-                    %% quicksort
                partition_mine(Smaller, Greater, E, Array),
                sort(SortedSmaller, Smaller),
                sort(SortedGreater, Greater),
                append(SortedArray, SortedSmaller, [E|SortedGreater]).

                %% partition(-Smaller, -Greater, +Elem, +Array): Partition the Array (of
numbers) into
                %%     subarrys Smaller and Greater than Elem.
partition_mine([], [], _, []).
partition_mine([Small|Smaller], Greater, Elem, [Small|Array])       :-
                Small le Elem,                      %% Small<= Elem,
                partition_mine(Smaller, Greater, Elem, Array).
partition_mine(Smaller, [Great|Greater], Elem, [Great|Array])       :-
                Great gt Elem,                      %% Great > Elem
                partition_mine(Smaller, Greater, Elem, Array).

                %% rotate(-RotatedList, +List, +N): Rotate the given List by N steps into
RotatedList.
rotate([], [], _).
rotate(RotatedList, List, N)    :-
                first_N_elems(FirstNElems, RestElems, List, N),
                append(RotatedList, RestElems, FirstNElems).

                %% first_N_elems(-FirstNElems, -RestElems, +List, +N): true if List =
FirstNElems + RestElems.
first_N_elems([], List, List, 0).
```

```
first_N_elems([], [], [], _N).
first_N_elems([A| NextElems], RestElems, [A| Rest], N)     :-
                int(N), N gt 0,
                first_N_elems(NextElems, RestElems, Rest, N.-.1).

                %% one_of(Elem, List): true if Elem is an element of the given List

%%one_of(Elem, List)                :-        %% this implementation has some disadvantages
%%              inlist(Elem, List).
one_of(Elem, [Elem| _]).
one_of(Elem, [_|List])             :-
                one_of(Elem, List).

                %% not_one_of(Elem, List): true if Elem is not an element of the given List
not_one_of(Elem, List)             :-
                outlist(Elem, List).

                %% an_elem_and_rest(Elem, Rest, List): true if Elem is an element of the List,
and Rest= List- Elem.
an_elem_and_rest(Elem, Rest, List)  :-
                an_elem_and_rest_aux(Elem, Rest, [], List).
an_elem_and_rest_aux(A, Rest, LeftList, [A| Remainder])  :-
                append(Rest, LeftList, Remainder).
an_elem_and_rest_aux(Elem, Rest, LeftList, [A|Remainder])          :-
                append(NewLeftList, LeftList, [A]),
                an_elem_and_rest_aux(Elem, Rest, NewLeftList, Remainder).


        %% random(-Elem, +List): return a random element (Elem) from the given list
random(Elem, List)    :-
                list(List),
                random_shuffle(ShuffledList, List),
                one_of(Elem, ShuffledList).

        %% random(-Random, +Range): return a random number in the range 0  ... Range
random(Rand, Range)           :-
                int(Range), Range> 0,
                random(R),
                modulo(R, Range+1, Rand).

        %% brandom(-BRandom): True if BRandom is a binary (i.e. 0 or 1) random number
brandom(BRand)                :-
                random(Rand),
                (((Rand/2147483647)>= 0.5) -> BRand = 1; BRand = 0).
```

```
        %% random(-Random): True if Random is a (pseudo-) random integer
        %%
        %% Note that this works because PrologIV can handle numbers greater than 32-bits
randomize(S)  :-        integer(S), record(seed, S).
random(X)     :-
        recorded(seed, S),
        (integer(S) -> S1 = S; S1= 1),
        Z= S1*16807,
        modulo(Z, 2147483647, X),
        record(seed, X).

%% KNJ: Seems like a very inadequate randomizer: we get numbers 7n+4 from it !!
%%              As such, you will have (for this randomizer): random()% 7= 4.
%%  --- Not used any more : KNJ ---
%%    KNJ: Modified the multiplier to a prime-no, and that seems to have improved the
generator.
%%%%%%%%%%
%  Random number generator from Pascal Bouvier of PrologIA :
%-- random.p4 --
% un generateur de nombres pseudo-aleatoires.
% A pseudo-random number generator.
% (formula got from C-ANSI random() ?)
%
% Pascal Bouvier, d'apres Prolog III
% (c) PrologIA 1996,1997

% initializer
orandomize(S) :- integer(S) ,!, record(oseed, S).
orandomize(S) :- var(S), record(oseed, 1).
orandom(X) :-
      recorded(oseed,S),
      (integer(S) -> S1 = S ; S1 = 1),
      Z = S1*1103515245 + 12345,
      modulo(Z, 65536*65536-1, Z1),
      record(oseed,Z1),
      modulo(Z,32767, X).


%%%%%%%%%%
```

```
' PrlgExpr.l
/*
 * PrlgExpr.l: Lexical analyzer for constraints:
 *               (splits constraints into lexical-components e.g. words, punctuations).
 *
*/
%{
#include <math.h>
#include <malloc.h>
#include <string.h>
#include "p4term.h"
#include "prlgHLAPI.h"
#include "PrlgExpr.tab.h"

extern int GetInString(char *buf, int max_size);
//extern int yylval;      -- supplanted by "extern YYSTYPE yylval;" in PrlgExpr.tab.h

#define YY_INPUT(buf, result_cnt, max_size)       {if (!(result_cnt= GetInString(buf,
max_size))) {buf[0]= YY_NULL; result_cnt= 1;} }

#define MAX_VAL_BUF                         64
static char valBuf[64][MAX_VAL_BUF];
static int valBuf_x= 0;
#define New_VAL_BUF              ((valBuf_x< MAX_VAL_BUF)? valBuf[valBuf_x++]:
(valBuf_x= 0, valBuf[valBuf_x++]))
//#define STRDUP(str)             (strcpy(New_VAL_BUF, (str)))
#define STRDUP(str)          (strdup(str))

%}

/*
 Note that, in flex, predefined char. classes (which must appear within [ ]) include:
UPPER                  [A-Z]       [:upper:]
LOWER                  [a-z]       [:lower:]
ALPHA                  [a-zA-Z]    [:alpha:]
ALNUM                  [A-Za-z0-9]  [:alnum:]
DIGIT          [0-9]      [:digit:]
SPACE                  [\\n\\r\\t\\v\\f\\O ]    [:space:]
*/

%%
"end_var_defs"                                   {yylval.ival= END_VAR_DEFS;
return(END_VAR_DEFS);}
"freeze"                                                  {yylval.ival= FREEZE;
return(FREEZE);}
```

```
"succeed"                                   {yylval.ival= SUCCEED;
return(SUCCEED);}
"fail"                                      {yylval.ival= FAIL;
return(FAIL);}
"if"                                        {yylval.ival= IF;
return(IF);}
"then"                                      {yylval.ival= THEN;
return(THEN);}
"else"                                      {yylval.ival= ELSE;
return(ELSE);}
"elseif"                                    {yylval.ival= ELSEIF;
return(ELSEIF);}
"int("                                      {yylval.ival= INT_PRED;
return(INT_PRED);}
"real("                                     {yylval.ival= REAL_PRED;
return(REAL_PRED);}
"fraction("                             {yylval.ival= FRACTION_PRED;
return(FRACTION_PRED);}
"list("                                     {yylval.ival= LIST_PRED;
return(LIST_PRED);}
"eq_vars("                              {yylval.ival= EQVARS_PRED;
return(EQVARS_PRED);}
"neq_vars("                             {yylval.ival= NEQVARS_PRED;
return(NEQVARS_PRED);}
"neq_varvals("                      {yylval.ival= NEQVARVALS_PRED;
return(NEQVARVALS_PRED);}
"optimizable_rel("          {yylval.ival= OPTIMIZABLEREL_PRED;
return(OPTIMIZABLEREL_PRED);}
"step"                                      {yylval.ival= STEP;
return(STEP);}
"symbol("                                   {yylval.ival=
SYMBOL_PRED; return(SYMBOL_PRED);}
"pi"                                            {yylval.fval=
(float)PI; return(PI);}
"in"                                            {yylval.ival=
IN_SET; return(IN_SET);}
"from"                                          {yylval.ival=
FROM_SET; return(FROM_SET);}
"notin"                                         {yylval.ival=
NOTIN_SET; return(NOTIN_SET);}
"not"                                       {yylval.ival= NOT;
return(NOT);}
"=="                                        {yylval.ival= EQ;
return(EQ);}
"=/="                                       {yylval.ival= NEQ;
```

```
return(NEQ);}
">="                                                        {yylval.ival= GE;
return(GE);}
"<="                                                        {yylval.ival= LE;
return(LE);}
"[!"                                                        {yylval.ival=
EXRANGE_START; return(EXRANGE_START);}
[[:upper:]_][[:alnum:]_]*                    {yylval.string= STRDUP(yytext);
return(VAR);}
[[:lower:]][[:alnum:]_]*                     {yylval.string= STRDUP(yytext);
return(ATOM_CONST);}
[[:digit:]]*"."[[:digit:]]+                  {yylval.string= STRDUP(yytext); /* stuffs
string-rep */ return(REALNUM);}
[[:digit:]]+                                      {yylval.ival= atoi(yytext);
return(INTNUM);}
[[:space:]]+                                      {continue;}
.                                                           {yylval.ival=
yytext[0]; return(yytext[0]);}
%%
int yywrap()  {return(1);}
```

```
' PrlgExpr.y
%{
/*
 * PrlgExpr.y: Parser for Prolog constraints  - to provide functionality for
 *             high-level communication, using mathematical expressions, between
 *             Prolog IV and other programs (e.g. TCA-GUI).
 *             (Use: bison -vd PrlgExpr.y to process it.)
 *
/

#include <stdio.h>
#include <time.h>
#include <string.h>
#include <malloc.h>
#include "p4term.h"
#include "prlgHLAPI.h"

#define CHECK_CNT                                          // <<-- define
it to check any buffer-overflows
#define VERSION           "3.3f"                    // <<-- update it appropriately
#define P4HLAPILIB                 "HLP4lib.p4"

#define TRUE              1
#define FALSE             0
#define PI_VAL            (3.14159265)
#define MAX(a, b)    ((a)>=(b)? (a):(b))
#define MIN(a, b)    ((a)<=(b)? (a):(b))
#define ABS(x)            ((x)>= 0? (x): -(x))
#define list(elem)   (cons((elem), NULL))
#define SAME(x, y)   (ABS((x)-(y))<= (Precision))

                            // sizes for some of P4 stacks in terms of cells (1 cell = 8
bytes)
#define P4HEAP_SIZE                      (1000000)                /*
P4-heap-stack-size in cell (= 8 bytes) counts [default: 700000] */
#define P4CHOICE_SIZE                (300000)                /*
P4-choice-stack-size in cell (= 8 bytes) counts [default: 50000] */

#define p4val_rational(T)    (p4val_as_double(T))
#define p4val_cstring(T)     (p4_symbol_to_cstring(p4val_symbol(T)))

#ifdef CHECK_CNT
static P4TERM check_term(P4TERM term)\
{if (!term) printf("\n***Received NULL P4TERM***\n");\
if (p4errno!= 0) {printf("\n***Unknown error occurred before the given term was
```

```
checked;***\n");fflush(stdout);}\
return term;}
#else
#define check_term(term)                    (term)
#endif /* CHECK_CNT */
#define P4MAKE_FUNC_0(func_str)
        check_term(p4make_atom(p4str2symbol(func_str)))
#define P4MAKE_FUNC_1(func_str, arg)
check_term(p4make_functor(1, p4str2symbol(func_str), (arg)))
#define P4MAKE_FUNC_2(func_str, arg1, arg2)            check_term(p4make_functor(2,
p4str2symbol(func_str), (arg1), (arg2)))
#define P4MAKE_FUNC_3(func_str, arg1, arg2, arg3)
check_term(p4make_functor(3, p4str2symbol(func_str), (arg1), (arg2), (arg3)))
#define P4MAKE_FUNC_4(func_str, arg1, arg2, arg3, arg4)
check_term(p4make_functor(4, p4str2symbol(func_str), (arg1), (arg2), (arg3), (arg4)))

#define P4AND(arg1, arg2)                                                 ((arg1) &&
(arg2)? P4MAKE_FUNC_2(",", arg1, arg2): (arg1)? (arg1): (arg2))
#define P4COMMA(arg1, arg2)                                               (P4AND(arg1,
arg2))
#define P4OR(arg1, arg2)                                                  ((arg1) &&
(arg2)? P4MAKE_FUNC_2(";", arg1, arg2): (arg1)? (arg1): (arg2))
#define P4EQ(arg1, arg2)                                                  ((arg1) &&
(arg2)? P4MAKE_FUNC_2("=", arg1, arg2): (arg1)? (arg1): (arg2))
#define P4NEQ(arg1, arg2)                                                 ((arg1) &&
(arg2)? P4MAKE_FUNC_2("dif", arg1, arg2): (arg1)? (arg1): (arg2))
#define P4IF_THEN(cond_t, then_t)                     (P4MAKE_FUNC_2("->", cond_t,
then_t))
#define P4IF_THEN_ELSE(cond_t, then_t, else_t)        (P4OR(P4IF_THEN(cond_t, then_t),
else_t))
#define P4IF_THEN_ELSEIF(then_cond_t, then_t, else_cond_t, else_t)
(P4OR(P4AND(then_cond_t, then_t), P4AND(else_cond_t, else_t)))
#define P4NOT(arg)
(P4MAKE_FUNC_1("\\+", arg))
#define P4CUT
                (P4MAKE_FUNC_0("!"))
#define P4TRUE
        (P4MAKE_FUNC_0("true"))
#define P4FAIL
        (P4MAKE_FUNC_0("fail"))
#define P4FALSE
        (P4FAIL)

        // need to wrap goal in p4call/1 before calling p4make_call() (to interpret +/2, -/2, ...)
#define P4MAKE_CALL(goal)      (p4make_call(P4MAKE_FUNC_1("p4call", goal)))
```

```
                    // due to a bug in p4what_is(), need to call dereference
#define P4WHAT_IS(term)          (p4what_is(dereference(term)))

          // max. size (in bytes) of an expression passed to the API
#define MAX_EXPR_SIZE          4096
          // max no. of variables (in one call to Prolog IV)
#define MAX_VAR_CNT                    1024
                         // max. length of variable name
#define MAX_VAR_NAME_LEN  32
                         // max-size of the funcTermBuf[]
#define MAX_FUNC_TERM_BUF_CNT             128
                         // max-size of the anonVarBuf[]
#define MAX_ANON_VAR_CNT          128
                         // max-size of the constBuf[]
#define MAX_CONST_CNT                  128
                         // max. arity of a functor
#define MAX_ARITY                      10

#define random                          rand                          // MS VC does not
have random() -- remove when we can have random()
#define srandom                         srand                         // MS VC does not
have srandom()-- remove when we can have random()
//extern void srandom(long);
//extern long random();
extern double atof();
//extern int isspace();
extern int p4errno;

#ifdef PRLGHLAPI
__declspec(dllexport) long AbortPrologSoln;         // Flag; if TRUE, Prolog
constraint-solving-process is aborted
#else
long AbortPrologSoln;          // Flag; if TRUE, Prolog constraint-solving-process is aborted
#endif // PRLGHLAPI

//typedef char BOOLEAN;

static struct s_rename_struct            // used in make_functor()
{
          char *func_name, *map_to_name;              // for mapping func-names e.g. gcd ->
gcdtemp
} FuncRenameList[]= {
                    {"ceiling", "ceil"},
                    {"gcd", "gcdtemp"},
                    {"lcm", "lcmtemp"},
```

```
		{"mod", "modtemp"},
		{"numden", "numdentemp"},
		{NULL, NULL}};	// make sure to end the list with {NULL, NULL}

typedef struct s_tabelem
{ // an element of the name-key table
 char type; // user-specified type (e.g. int(X)) of the variable; 0 if any type will do.
	char ongrid;	// flag: true if var-value is to be an integral multiple of its precision; false
otherwise
 double precision;		// precision for the variable
 char name[MAX_VAR_NAME_LEN+1]; // variable name
 P4TERM term;	// P4 Prolog term representation for the variable
 struct s_tabelem *next;
	char is_independent_var;		// TRUE if the variable is independent (& is not a
constant); FALSE otherwise
} TabElem;

typedef struct s_constant
{
	P4TERM term;
	Value value;
} Const;

static char inExprBuf[MAX_EXPR_SIZE+1]= {0};		// buffer to store the incoming
expression (string) in - for parsing purposes
static int inExprBuf_x= 0, inExprBuf_cnt= 0;
#define INIT_inExprBuf			{inExprBuf_cnt= inExprBuf_x= 0;}

		// buffer to store func-terms for arity conversion (i.e. X= func(Y, Z). -> X= _R,
func(_R, Y, Z).)
static P4TERM funcTermBuf[MAX_FUNC_TERM_BUF_CNT]= {NULL};
static int funcTermBuf_x= 0;
#define INIT_funcTermBuf		{funcTermBuf_x= 0; memset(funcTermBuf, 0,
MAX_FUNC_TERM_BUF_CNT* sizeof(P4TERM));}
#ifdef CHECK_CNT
#define ADD_funcTermBuf(term)	{if (funcTermBuf_x< MAX_FUNC_TERM_BUF_CNT)
funcTermBuf[funcTermBuf_x++]= (term); else {printf("\n***FUNC_TERM_BUF
overflow***\n");fflush(stdout);}}
#else
#define ADD_funcTermBuf(term)	{funcTermBuf[funcTermBuf_x++]= (term);}
#endif /* CHECK_CNT */
#define FOR_EACH_FUNC_TERM(term)		{int _i_func; for(_i_func= 0; _i_func<
funcTermBuf_x && ((term)= funcTermBuf[_i_func]); _i_func++) {
#define END_FOR_EACH_FUNC_TERM(term)	}}
```

```
                    // buffer for anonymous var's used in the functions
static P4TERM anonVarBuf[MAX_ANON_VAR_CNT]= {NULL};
static int anonVarBuf_x= 0;
#define INIT_anonVarBuf                         {anonVarBuf_x= 0; memset(anonVarBuf, 0,
MAX_ANON_VAR_CNT* sizeof(P4TERM));}
#ifdef CHECK_CNT
#define ADD_anonVarBuf(anon_term)       {if (anonVarBuf_x< MAX_ANON_VAR_CNT)
anonVarBuf[anonVarBuf_x++]= (anon_term); else {printf("\n***ANON_VAR_BUF
overflow***\n");fflush(stdout);}}
#else
#define ADD_anonVarBuf(anon_term) {anonVarBuf[anonVarBuf_x++]= (anon_term);}
#endif /* CHECK_CNT */
#define FOR_EACH_ANON_VAR(anon_term)      {int _i_anon; for(_i_anon= 0; (_i_anon<
anonVarBuf_x) && ((anon_term)= anonVarBuf[_i_anon]); _i_anon++) {
#define END_FOR_EACH_ANON_VAR(anon_term) }}

                    // buffer to store number-constant's (with their values)
static Const constBuf[MAX_CONST_CNT]= {0};
static int constBuf_x= 0;
#define INIT_constBuf                                        {constBuf_x= 0;}
#ifdef CHECK_CNT
#define ADD_constBuf(trm, const_val)       {if (constBuf_x< MAX_CONST_CNT)
{constBuf[constBuf_x].term= (trm); constBuf[constBuf_x].value= (const_val);constBuf_x++;}
else {printf("\n***CONST_BUF overflow***\n");fflush(stdout);}}
#else
#define ADD_constBuf(trm, const_val)       {constBuf[constBuf_x].term= (trm);
constBuf[constBuf_x].value= (const_val); constBuf_x++;}
#endif /* CHECK_CNT */

                    // setup to allocate TabElem's from a circular buffer - quick, easy to reinitialize,
& no need to free up pointers
static TabElem tabSpace[MAX_VAR_CNT]= {0};    // space to alloc TabElem from
static int tabSpace_x= 0;
                    // alloc a new TabElem from a circular buffer
#ifdef CHECK_CNT
#define NEW_TAB_ELEM   ((tabSpace_x< MAX_VAR_CNT)? &tabSpace[tabSpace_x++]:
(tabSpace_x= 0, printf("\n***tabSpace-buffer overflow***\n"), fflush(stdout),
&tabSpace[tabSpace_x++]))
#else
#define NEW_TAB_ELEM   ((tabSpace_x< MAX_VAR_CNT)? &tabSpace[tabSpace_x++]:
(tabSpace_x= 0, &tabSpace[tabSpace_x++]))
#endif /* CHECK_CNT */
                    // [re]intialize the table
#define INIT_tabSpace                   {tabSpace_x= 0; memset(tabSpace, 0,
MAX_VAR_CNT*sizeof(TabElem));}
```

```
// [hash] table for Var's
#define VAR_TABLE_SIZE          53
static TabElem *varTable[VAR_TABLE_SIZE] = {NULL};

// varList: array of var's from the current constraint e.g. X, Y from "X= Y+2, Y=6."
static TabElem *varList[MAX_VAR_CNT]= {NULL};
static int varList_x= 0;
// add the given TabElem-ptr to the varList
#ifdef CHECK_CNT
#define ADD_varList(p_tabElem)     {if (varList_x< MAX_VAR_CNT) varList[varList_x++]= (p_tabElem); else {printf("\n***varList-buffer overflow***\n");fflush(stdout);}}
#else
#define ADD_varList(p_tabElem)     {varList[varList_x++]= (p_tabElem);}
#endif /* CHECK_CNT */
// [re]intialize the varList
#define INIT_varList  {varList_x= 0;}
#define var_CNT                          (varList_x)

// initialize all the variable-related space
#define INIT_vars                  {memset(varTable, 0, VAR_TABLE_SIZE*sizeof(TabElem *)); INIT_varList; INIT_tabSpace;}
// to do something for each var in current constraint; sets p_tab_elem & its index in varList
#define FOR_EACH_VAR(p_tab_elem, x)          {int _i_var; for(_i_var= 0; ((x)=_i_var)< var_CNT && ((p_tab_elem)= varList[_i_var]); _i_var++) {
#define END_FOR_EACH_VAR(p_tab_elem, x)    }}

// buffer for return values
#define MAX_VAL_BUF_LEN               (1024)
static Value valBuf[MAX_VAL_BUF_LEN];
static int valBuf_x= 0;
#ifdef CHECK_CNT
#define NEW_VALUE                      ((valBuf_x< MAX_VAL_BUF_LEN)? &valBuf[valBuf_x++]: (valBuf_x= 0, printf("\n***valBuf-buffer overflow***\n"), fflush(stdout), &valBuf[valBuf_x++]))
#else
#define NEW_VALUE                      ((valBuf_x< MAX_VAL_BUF_LEN)? &valBuf[valBuf_x++]: (valBuf_x= 0, &valBuf[valBuf_x++]))
#endif /* CHECK_CNT */
#define INIT_valBuf            {valBuf_x= 0;}

// buffer for enumerated-range terms
#define MAX_ENUM_RANGE_TERMS     (128)
static P4TERM enumRangeTermBuf[MAX_ENUM_RANGE_TERMS]= {NULL};
```

```
static int enumRangeTermBuf_x= 0;
#define enumRangeTermBuf_CNT            (enumRangeTermBuf_x)
#define INIT_enumRangeTermBuf   {enumRangeTermBuf_x= 0;}
#ifdef CHECK_CNT
#define ADD_ENUM_RANGE_TERM(term)       {if (enumRangeTermBuf_x<
MAX_ENUM_RANGE_TERMS) enumRangeTermBuf[enumRangeTermBuf_x++]= (term);
else {printf("\n***enumRangeTermBuf overflow***\n");fflush(stdout);}}
#else
#define ADD_ENUM_RANGE_TERM(term)
{enumRangeTermBuf[enumRangeTermBuf_x++]= (term);}
#endif /* CHECK_CNT */
#define ENUM_RANGE_TERM(x)              (((x)< enumRangeTermBuf_CNT) &&
((x)>= 0)? enumRangeTermBuf[x]: NULL)
#define FOR_EACH_ENUM_RANGE_TERM(term, x)     {int _i_vrange; for(_i_vrange= 0;
(((x)=_i_vrange)< enumRangeTermBuf_x) && ((term)= enumRangeTermBuf[_i_vrange]);
_i_vrange++) {
#define END_FOR_EACH_ENUM_RANGE_TERM(term, x)      }}
                    // swap the positions of the terms (given by their indices in the buffer) in
the variable-range buffer
#define SWAP_ENUM_RANGE_TERMS(term_x, term_y)        {P4TERM _t_vrange;\
if (((term_x)< enumRangeTermBuf_CNT) && ((term_y)< enumRangeTermBuf_CNT)) {\
        _t_vrange= enumRangeTermBuf[term_x]; enumRangeTermBuf[term_x]=
enumRangeTermBuf[term_y];\
        enumRangeTermBuf[term_x]= _t_vrange;\
        }}

                // buffer for var-type (e.g. int(X); eq_vars(X, Y); neq_vars(X, Y,Z) ) terms
#define MAX_VAR_TYPES_TERMS       (128)
static P4TERM varTypesTermBuf[MAX_VAR_TYPES_TERMS]= {NULL};
static int varTypesTermBuf_x= 0;
#define varTypesTermBuf_CNT             (varTypesTermBuf_x)
#define INIT_varTypesTermBuf      {varTypesTermBuf_x= 0;}
#ifdef CHECK_CNT
#define ADD_VAR_TYPES_TERM(term)  {if (varTypesTermBuf_x<
MAX_VAR_TYPES_TERMS) varTypesTermBuf[varTypesTermBuf_x++]= (term); else
{printf("\n***varTypesTermBuf overflow***\n");fflush(stdout);}}
#else
#define ADD_VAR_TYPES_TERM(term)  {varTypesTermBuf[varTypesTermBuf_x++]=
(term);}
#endif /* CHECK_CNT */

                // buffer for storing solutions so we can return solutions in some (e.g.
breadth-first) order
#define MAX_SOLN_BUF_LEN              (1024)
                // the soln-buffer really is an array of vectors of solutions (e.g. [[x1,y1],[x2,y2],...]
```

```
	//	(we take the vector-width to be the no. of variables in the constraint==
varList_x)
static Value solnBuf[MAX_SOLN_BUF_LEN];
static Value tmpsolnBuf[MAX_SOLN_BUF_LEN/2]; // tmp solution buffer - useful for
shuffling solutions
static int solnVec_x= 0, solnCnt= 0, curSoln_x= 0, doBufferSoln= FALSE;
#define INIT_solnBuf		{solnVec_x= 0; solnCnt= 0; curSoln_x= 0; doBufferSoln=
FALSE;}
#define NEW_SOLN_VECTOR			((var_CNT<= 0)? NULL:\
						((solnVec_x<
(int)(MAX_SOLN_BUF_LEN/var_CNT))? &solnBuf[var_CNT*solnVec_x++]: NULL))
#define SOLN_VECTOR_SIZE		(var_CNT*sizeof(Value))
#define SET_BUF_SOLN_CNT(soln_cnt)		{solnCnt= (soln_cnt);}
	// ptr to the value (by its index from the varList) of a variable in specified solution
(by its index) in the solnBuf
#define p_VAR_VALUE_in_solnBuf(soln_x, var_x)	(((soln_x)>= solnCnt)? NULL:
(((var_x)< var_CNT)? &solnBuf[var_CNT*(soln_x)+ (var_x)]: NULL))
	// ptr to the value (by its index from the varList) of a variable in current-solution
#define CUR_BUF_SOLUTION(var_x)		p_VAR_VALUE_in_solnBuf(curSoln_x,
var_x)
#define NEXT_BUF_SOLUTION			((curSoln_x>= (solnCnt-1))? NULL:
&solnBuf[varList_x*++curSoln_x])
#define FOR_EACH_SOLN_VECTOR(p_soln_vec, x)	{int _i_solnx; for(_i_solnx= 0;
(((x)=_i_solnx)< solnCnt) && ((p_soln_vec)= &solnBuf[var_CNT*_i_solnx]); _i_solnx++) {
#define END_FOR_EACH_SOLN_VECTOR(p_soln_vec, x)	}}
	// swap the specified (by their indices) solutions (value-vectors) in the solnBuf
#define SWAP_SOLUTIONS(soln_i, soln_j)\
if ((soln_i)< solnCnt && (soln_j)< solnCnt)\
 {\
 Value __tmp; int __x;\
 for(__x= 0; __x< var_CNT; __x++)\
	{\
	__tmp= *p_VAR_VALUE_in_solnBuf(soln_i, __x);\
	*p_VAR_VALUE_in_solnBuf(soln_i, __x)=
*p_VAR_VALUE_in_solnBuf(soln_j, __x);\
	*p_VAR_VALUE_in_solnBuf(soln_j, __x)= __tmp;\
	}\
 }
	// move the specified (by its from & to-index) solution in the solnBuf
#define MOVE_SOLUTION(to_soln_x, from_soln_x)\
if ((to_soln_x)< solnCnt && (from_soln_x)< solnCnt)\
 {\
 int __x;\
 for(__x= 0; __x< var_CNT; __x++)\
	{\
```

```
		*p_VAR_VALUE_in_solnBuf(to_soln_x, __x)=
*p_VAR_VALUE_in_solnBuf(from_soln_x, __x);\
		}\
 }
		// copy the specified (by its index) solution from solnBuf into the tmpsolnBuf
#define COPY_SOLN_to_TMPBUF(soln_i)\
if ((soln_i)< solnCnt)\
 {\
 int __x;\
 for(__x= 0; __x< var_CNT; __x++)\
		{\
		tmpsolnBuf[varList_x*(soln_i)+ __x]= *p_VAR_VALUE_in_solnBuf(soln_i,
__x);\
		}\
 }
		// copy the specified (by its from & to-index) solution from tmpsolnBuf into the
solnBuf
#define COPY_SOLN_from_TMPBUF(to_soln_x, from_tmp_soln_x)\
if ((to_soln_x)< solnCnt && (from_tmp_soln_x)< solnCnt)\
 {\
 int __x;\
 for(__x= 0; __x< var_CNT; __x++)\
		{\
		*p_VAR_VALUE_in_solnBuf(to_soln_x, __x)=
tmpsolnBuf[varList_x*(from_tmp_soln_x)+ __x];\
		}\
 }


		// misc. static var's
static int semError = 0;				// set to non-zero in case of semantic error.
static BOOLEAN calledStartProlog4 = FALSE;
static BOOLEAN startedNewProlog4Session= FALSE;
static int resultFromProlog4= 0;
static BOOLEAN fractionalizeRational= FALSE;		// if TRUE, fractionalize all rationals
static BOOLEAN roundoffFractionOnlyRational= TRUE;	// if TRUE round-off fraction-only
rationals (e.g. 2/3); otherwise, return fraction-only rationals as fractions
static char *p4Argv[3]= {"P4LIB", "-banner=off", NULL};		// init-args to Prolog
IV
static BOOLEAN useIntervalSolver= FALSE;
static char curConstraint[MAX_EXPR_SIZE];
static double Precision= DEF_PRECISION;
static BOOLEAN RandomizeConstraints= FALSE;
static int SolnDiffWt = DEF_SOLN_DIFF_WT;			// weight to indicate how
"different" the solns must be from each other
```

```
static long BSeed = 1;                    // seed for the random-bit-generator: brandom()
static BOOLEAN enumerateVarsRandomlyNoHistory = FALSE;   // True if independent vars are to be enumerated randomly (without keeping track of their past values)
static int TimedThreadCount= 0;      // keeps a running count of active threads
static HANDLE TimedSolnMutex; // a semaphore to access TimedThreadCount between threads
static BOOLEAN AbortConstraintTimer= FALSE; // a flag to abort constraint-timer
static BOOLEAN AbortWatchPrologThread = FALSE; // a flag to abort WatchAbortPrologSoln thread
static char *PrologSolnInterruptFile= NULL;        // Presence of this file indicates interruption of Prolog-solution

                // misc. static declarations
static BOOLEAN init_solve_constraint(int keep_solns);
static TabElem *get_var(char *var);
static P4TERM get_var_term(char *var);
static Value *get_term_value(P4TERM term);
static List *cons(void *elem, List *lst);
static List *ncons(List *lst, void *elem);
static P4SYMBOL p4str2symbol(char *str);   // pseudo p4-routine
static P4TERM p4make_atom_from_cstring(char *str);          // pseudo p4-routine
static P4TERM P4Make_Rational(double val);      // pseudo p4-routine
static int p4is_constant(P4TERM term);
static Value * get_var_value(char *var);
static P4TERM get_value_term(Value *val);
static int auxSolveConstraint(char *constraint, int keep_prev_soln, long msec);
static int h_auxSolveConstraint(char *constraint, int keep_prev_soln, int enum_vars_randomly, long msec);
static P4TERM make_termlist2term(List *lst);
static P4TERM make_valslist2term(List *lst);
static P4TERM combine_terms_array(P4TERM terms[], long size_terms, int do_conjunct, int randomize, P4TERM var, int var_eq);
static P4TERM combine_terms_list(List *terms_lst, int do_conjunct, int randomize, P4TERM var, int var_eq);
static int auxSolveConstraintOrdered(char *constraint, int order_type, int max_soln);
static double align_val_with_precision(double val, double precision);
static TabElem *get_term_tabelem(P4TERM term);
static int getTimedThreadCount();

                // misc. extern declarations
extern int yyparse(void);
extern int yyerror(char *s);
extern int yylex(void);
```

```
#ifndef PRLGHLAPI
#include "cmn_drvr.c"                 // driver to test all the stuff out
#endif /* PRLGHLAPI */


                // public functions

char * CCONV GetHLAPIVersion()
{// return the current version of the Prolog HL API
return(VERSION);
}

BSTR CCONV VBGetHLAPIVersion()      // wrapper to GetHLAPIVersion() for VB
{
char *c_ver;
BSTR vb_ver;

c_ver= GetHLAPIVersion();
vb_ver= SysAllocStringByteLen(NULL, strlen(c_ver)+1);  // alloc a new BSTR
strcpy((char *)vb_ver, c_ver);

return(vb_ver);
}

DWORD WINAPI watchAbortPrologSoln(void *null)
{       // watch AbortPrologSoln-variable or the presence-of-Interrupt-file; if either condition
becomes true, then abort Prolog constraint-solving-process
#define SLEEP_INTERVAL          (2000) // in msec
extern int access();

for(AbortWatchPrologThread= FALSE; !AbortWatchPrologThread;
Sleep(SLEEP_INTERVAL)) // run until someone turns AbortWatchPrologThread to TRUE
        {
        if (AbortPrologSoln || (PrologSolnInterruptFile && (access(PrologSolnInterruptFile,
0)==0)))
                {
                prolog_events |= (1L<< 16);
                }
        }
AbortWatchPrologThread= FALSE;  // set if FALSE here as an indicator that this thread is
finished

return(0);
}
```

```
int CCONV SetPrologInterruptFile(char *interrupt_filename)
{       // set the filename whose presence interrupts Prolog-solution
ProlognSolnInterruptFile= interrupt_filename? strdup(interrupt_filename): NULL;

return(TRUE);
}

static int StartProlog4Session_aux(char *p4hlapilib_file, long heapsize, long choicesize)
{       // starts Prolog IV, return true (1) if ok, false (0) otherwise.
        // p4hlapilib_file is the pathname to the high-level Prolog IV API library file
        //      heapsize is the heap-stack size; choicesize is the choice-stack size.
        //      ((argc, argv[]) are arguments to Prolog IV.)
int status, argc;
char *p4argv[32]= {"P4LIB", NULL};                  // init-args to Prolog IV
char *banner_arg= "-banner=off";
char heapsize_str[32], choicesize_str[32];
P4TERM term;
long dummy, id;

if (!(TimedSolnMutex= CreateMutex(NULL, FALSE, NULL)))    // create a semaphore to
access TimedThreadCount
        return(FALSE);

AbortPrologSoln= FALSE;
if (!CreateThread(NULL, 0, watchAbortPrologSoln, &dummy, 0, &id))
        {
        printf("Unable to create watch-thread\n");
        return(FALSE);
        }

if (!calledStartProlog4)
        {
        calledStartProlog4 = TRUE;
        p4errno = 0;            // reset PrologIV error-no
                        // format arguments for the P4 commandline
        argc= 1;
        p4argv[argc++]= banner_arg;
        p4argv[argc++]= "-heap";
        sprintf(heapsize_str, "%d", heapsize);
        p4argv[argc++]= heapsize_str;
        p4argv[argc++]= "-choice";
        sprintf(choicesize_str, "%d", choicesize);
        p4argv[argc++]= choicesize_str;

        p4argv[argc]= NULL;
```

```
		status= (p4init(argc, p4argv)==0);
		if (!status)
			return(FALSE);

		srandom(time(NULL));
		if (p4hlapilib_file && !Compile(p4hlapilib_file))			// load High-level
	PrologIV library
			return(FALSE);

			// set the randomizer-seed in Prolog - from time(NULL)
		p4new_session();
		term= P4MAKE_FUNC_1("randomize", p4make_lint(time(NULL)));
		p4make_call(term);
		status = p4next_solution();
		if (status==P4SESSION_ERROR)
			return (FALSE);
		p4finish_session();

	#ifdef DO_SETOF_INIT
		// we don't use setof/bagof for now - so, no need to do this initialization
		// initialization to get around a bug in P4 when calling setof/3 or bagof/3
		p4new_session();
		term= P4AND(P4MAKE_FUNC_2("def_array",
	p4make_atom_from_cstring("tab_bagof"), p4make_lint(100)),
					P4AND(P4MAKE_FUNC_2("record",
	p4make_atom_from_cstring("block_limit"), p4make_lint(0)),
						P4MAKE_FUNC_2("record",
	p4make_atom_from_cstring("last_bag_bound"), p4make_lint(0))));
		p4make_call(term);
		status = p4next_solution();
		p4finish_session();
		// end-of-initialization to get around a bug in P4 when calling setof/3 or bagof/3
	#endif /* DO_SETOF_INIT */
		}

	return(TRUE);
	}

	int CCONV StartProlog4Session(char *p4hlapilib_file)
	{	// starts Prolog IV, return true (1) if ok, false (0) otherwise.
		// p4hlapilib_file is the pathname to the high-level Prolog IV API library file

	return(StartProlog4Session_aux(p4hlapilib_file, P4HEAP_SIZE, P4CHOICE_SIZE));
	}
```

```
int CCONV StartProlog4SessionSetStacks(char *p4hlapilib_file, long heapsize, long choicesize)
{       // starts Prolog IV, return true (1) if ok, false (0) otherwise.
        // p4hlapilib_file is the pathname to the high-level Prolog IV API library file
        //      heapsize is the heap-stack size; choicesize is the choice-stack size.

return(StartProlog4Session_aux(p4hlapilib_file, MAX(heapsize,P4HEAP_SIZE),
MAX(choicesize, P4CHOICE_SIZE)));
}

int CCONV StartProlog4SessionDefault()
{       // starts Prolog IV with default parameters, return true (1) if ok, false (0) otherwise.

return(StartProlog4Session_aux("c:\\MyDLL\\HLP4lib.p4", P4HEAP_SIZE,
P4CHOICE_SIZE));
}

int CCONV EndProlog4Session()
{       // end Prolog IV session; return true (1) if ok, false (0) otherwise.
                // Abort all the threads: the WatchAbortPrologSoln, and constraint-timer
AbortWatchPrologThread= TRUE;                   // abort WatchAbortPrologSoln thread
AbortConstraintTimer= TRUE;      // Abort all previous constraint-timers
while(getTimedThreadCount()> 0)  // Sleep until all constraint-timer threads are
finished/aborted
        Sleep(5);

while(AbortWatchPrologThread)    // Sleep until WatchAbortPrologSoln thread is
finished/aborted
        Sleep(5);

return(TRUE);
}

static char brandom()
// return a random-bit (0 or 1) (uses BSeed);
//      uses primitive polynomial modulo 2 (PPM2) of degree 18
{
unsigned char newbit;

newbit = (BSeed>> 17) & 1          // bit 18
                    ^ (BSeed>> 4) & 1
                    ^ (BSeed>> 1) & 1
                    ^ (BSeed & 1);
BSeed = (BSeed<< 1) | newbit;               // shift bits; put newbit at end

return(newbit);
```

```
}

static double align_val_with_precision(double val, double precision)
// align the given val with the given precision; return the aligned value.
// (That is, returned-val= N* precision, (where N is integer) such that |returned-val - val| is
minimum
{
int N;

if (precision== 0)
        return(val);

N= (val>= 0)? (int)((val/precision)+0.5): (int)((val/precision)- 0.5);
val= N* precision;

return(val);
}
                // start-of Thread-related functions
static int incTimedThreadCount()
{       // increment the TimedThreadCount; (inside a semaphore because of multithreaded
access.)
WaitForSingleObject(TimedSolnMutex, INFINITE);
TimedThreadCount++;
ReleaseMutex(TimedSolnMutex);

return(0);
}

static int decTimedThreadCount()
{       // decrement the TimedThreadCount; (inside a semaphore because of multithreaded
access.)
WaitForSingleObject(TimedSolnMutex, INFINITE);
TimedThreadCount--;
ReleaseMutex(TimedSolnMutex);

return(0);
}

static int getTimedThreadCount()
{       // return the current TimedThreadCount; (inside a semaphore because of multithreaded
access.)
int cnt;

WaitForSingleObject(TimedSolnMutex, INFINITE);
cnt= TimedThreadCount;
```

```
ReleaseMutex(TimedSolnMutex);

return(cnt);
}

DWORD WINAPI theConstraintTimer(void *p_msec)
{
long msec= *((long *)p_msec);

// printf("Going to sleep in the timer\n"); // -- debug

if (msec> 0)
        {
        incTimedThreadCount();      // increment the active-thread count
        if (msec<= 1000)
                Sleep(msec);
        else
                {       // check AbortConstraintTimer frequently between sleeps
                long t;
                for(t=0; (t< msec) && !AbortConstraintTimer; t+= 1000)
                        Sleep(1000);
                }
        }
else
        return(0);

// printf("Out of sleep in the timer ... setting prolog_events\n"); // -- debug

prolog_events |= (1L<< 16);

decTimedThreadCount();      // decrement the active-thread count
return(0);
}
                // end-of Thread-related functions

static int auxSolveConstraint(char *constraint, int keep_prev_solns, long msec)
{       // solve the given constraint (e.g. "X= Y+ 4, Y=2.");
        // backtracks over the previous solution if the constraint is empty (i.e. NULL)
        // if keep_prev_solns is true, do not discard the previous solutions
        // if msec> 0, the solver exits in the given max. time (ms) - whether the constraint is
solved or not
        // return true (=1) [false (=0)] if the constraint is [un]solvable;
        //      returns negative integer in error (e.g. if the constraint could not be parsed, or
solver aborted).
int stat;
```

```
long id;

stat     = 0;
        // reset all Prolog flags/events
resultFromProlog4= 0;
AbortPrologSoln= FALSE;
prolog_events= 0;

if (msec> 0)
        {       // the solution-process must be time-limited
        AbortConstraintTimer= TRUE;         // Abort all previous constraint-timers
        while(getTimedThreadCount()> 0)    // Sleep until all previous active threads are
finished/aborted
                Sleep(5);
        AbortConstraintTimer= FALSE;

        prolog_events= 0;      // reset all previous Prolog events
        if (!CreateThread(NULL, 0, theConstraintTimer, &msec, 0, &id))
                {
                printf("Unable to create thread\n");
                return(ERR_UNABLE_TO_CREATE_THREAD);
                }
        }

if (constraint)
 {      // a new constraint given
 semError= 0;
 if (startedNewProlog4Session) // a goal called previously - end that.
                p4finish_session();
          // initialize all the tables, spaces, counts;
 if (!init_solve_constraint(keep_prev_solns))
                return(ERR_INITIALIZATION);
                        // start a new Prolog session

 p4new_session();
 startedNewProlog4Session= TRUE;

 if ((inExprBuf_cnt= strlen(constraint))> MAX_EXPR_SIZE)
        return(ERR_CONSTRAINT_TOO_LONG);

 strcpy(inExprBuf, constraint);
 inExprBuf_x= 0;

 stat= yyparse();
 }
```

```
else if (calledStartProlog4)     // backtrack over the previous result
 {
        if (semError==0)
                resultFromProlog4= p4next_solution();
        else
                return (FALSE);
 }

if (p4errno!= 0)
        {
        printf("\n***Unknown error from PrologIV in auxSolveConstraint()***\n");
        return (ERR_PARSE);
        }

if (semError== 0 && stat== 0)
 return((resultFromProlog4== P4SESSION_SOLUTION)? TRUE:
                (resultFromProlog4== P4SESSION_END)? FALSE:
                        (prolog_events!= 0)? ERR_SOLN_INTERRUPTED:
ERR_PROLOG_SOLVER);
else if (semError!= 0)
 return(semError);
else
 return(ERR_PARSE);
}

static int h_auxSolveConstraint(char *constraint, int keep_prev_solns, int enum_vars_randomly,
long msec)
{       // solve the given constraint (e.g. "X= Y+ 4, Y=2."); using a linear/interval solver as
needed
        // backtracks over the previous solution if the constraint is empty (i.e. NULL)
        // if keep_prev_solns is true, do not discard the previous solutions
        //      if enum_vars_randomly is true, enumerate independent vars randomly, without
keepingtrack of their previous values
        // if msec> 0, the solver exits in the given max. time (ms) - whether the constraint is
solved or not
        //                      >>(we LOSE track of unique solutions (-cnt) when
enum_vars_randomly is true.)<<
        // return true (=1) [false (=0)] if the constraint is [un]solvable;
        //      returns negative integer in error (e.g. if the constraint could not be parsed).
int stat;

if (constraint)
 {
 if (strlen(constraint)> MAX_EXPR_SIZE)
        return(ERR_CONSTRAINT_TOO_LONG);
```

```
 strcpy(curConstraint, constraint);
 useIntervalSolver= FALSE;
 enumerateVarsRandomlyNoHistory= enum_vars_randomly;

 if ((stat=auxSolveConstraint(curConstraint, keep_prev_solns, msec))==0 || (stat> 0 &&
!IsFullyConstrained(NULL)))
            {
            useIntervalSolver= TRUE;
            return(auxSolveConstraint(curConstraint, keep_prev_solns, msec));
            }
 return(stat);
 }
else
 return(auxSolveConstraint(NULL, keep_prev_solns, msec));
}

int CCONV SolveConstraint(char *constraint)
{       // solve the given constraint (e.g. "X= Y+ 4, Y=2."); using a linear/interval solver as
needed
        //  backtracks over the previous solution if the constraint is empty (i.e. NULL)
        // return true (=1) [false (=0)] if the constraint is [un]solvable;
        //       returns negative integer in error (e.g. if the constraint could not be parsed).
return(h_auxSolveConstraint(constraint, FALSE, FALSE, -1));
}

int CCONV SolveConstraintRandomly(char *constraint)
{       // random-solve the given constraint (e.g. "X= Y+ 4, Y=2."); using a linear/interval solver
as needed
        //  backtracks over the previous solution if the constraint is empty (i.e. NULL)
        //       >>      NO track of unique solutions (-cnt) is kept : You may not get unique
solutions <<
        // return true (=1) [false (=0)] if the constraint is [un]solvable;
        //       returns negative integer in error (e.g. if the constraint could not be parsed).

return(h_auxSolveConstraint(constraint, FALSE, TRUE, -1));
}

int CCONV TimedSolveConstraint(char *constraint, long msec)
{       // solve the given constraint (e.g. "X= Y+ 4, Y=2."); using a linear/interval solver as
needed
        //  backtracks over the previous solution if the constraint is empty (i.e. NULL)
        // ensures that the call finishes in the given time (ms) - whether the constraint is solved or
not
        // return true (=1) [false (=0)] if the constraint is [un]solvable;
        //       returns negative integer in error (e.g. if the constraint could not be parsed, or
```

```
could not be solved in given time).

return(h_auxSolveConstraint(constraint, FALSE, FALSE, msec));
}

int CCONV TimedSolveConstraintRandomly(char *constraint, long msec)
{       // random-solve the given constraint (e.g. "X= Y+ 4, Y=2."); using a linear/interval solver
as needed
        // backtracks over the previous solution if the constraint is empty (i.e. NULL)
        //      >>      NO track of unique solutions (-cnt) is kept : You may not get unique
solutions <<
        // return true (=1) [false (=0)] if the constraint is [un]solvable;
        //      returns negative integer in error (e.g. if the constraint could not be parsed).

return(h_auxSolveConstraint(constraint, FALSE, TRUE, msec));
}

static int solution_cmp(const Value *vec1, const Value *vec2)
{ // compare the two solution-vectors; return >,=,< 0 as vec1 >,=,< vec2.
int i, wt, dist;
double val1, val2;

for(i= dist= 0; i< varList_x; i++)
  {
  wt= varList_x- i;
  val1= (vec1[i].type== VAL_INTEGER)? vec1[i].value.integer: (vec1[i].type==
VAL_IRRATIONAL)? (vec1[i].value.real.lower.val+vec1[i].value.real.upper.val)/2:
              (vec1[i].type== VAL_REAL || vec1[i].type== VAL_RATIONAL_FLOAT ||
vec1[i].type== VAL_RATIONAL_FRACTION)? vec1[i].value.rational.real: 0;
  val2= (vec2[i].type== VAL_INTEGER)? vec2[i].value.integer: (vec2[i].type==
VAL_IRRATIONAL)? (vec2[i].value.real.lower.val+vec2[i].value.real.upper.val)/2:
              (vec2[i].type== VAL_REAL || vec2[i].type== VAL_RATIONAL_FLOAT ||
vec2[i].type== VAL_RATIONAL_FRACTION)? vec2[i].value.rational.real: 0;
  dist+= (int)(wt*(val1- val2));        // weighted distance
  }

return(dist);
}

static int auxSolveConstraintOrdered(char *constraint, int order_type, int max_soln)
{       // solve the given constraint (e.g. "X= Y+ 4, Y=2."); using a linear/interval solver as
needed
        // solve to find the given no. (= max_soln) of solutions if max_soln is positive; find
(nearly) all solutions if it is negative
        // backtracks over the previous solution if the constraint is empty (i.e. NULL)
```

```
        // Store all the solutions in a buffer, order them (by the given order_type)so we can return solutions in an ordered fashion
        // present the solutions conforming to the given order (e.g. ORDER_DIFF_TOGETHER)
        // returns, on first call (i.e. when constraint is non-NULL), (1+ the_total_count_of_solutions) (> 0) [false (=0)] if the constraint is [un]solvable;
        //              (note that in case of constraints without variables (e.g. "4= 4."), total-no.-of-solutions is 0, though the constraint is provable.)
        // returns, on subsequent calls (i.e. when constraint is NULL), true (= 1) [false (=0)] if a solution exists [does not exist];
        //      returns negative integer in error (e.g. if the constraint could not be parsed).
        //       (The P4 setof/3 & bagof/3 are buggy - hence we simulate them here ourselves.)
int i, iy, soln_cnt, half_cnt, stat;
Value *p_vec, *pval;
TabElem *p_tab_elem;
int keep_prev_solns;
extern void qsort();

keep_prev_solns= FALSE;
RandomizeConstraints= FALSE;
if (constraint)
  {
  for(soln_cnt= 0; ((max_soln< 0) || (soln_cnt< max_soln)) && ((stat= h_auxSolveConstraint(constraint, keep_prev_solns, FALSE,-1))> 0); )
        {
        RandomizeConstraints= (order_type==ORDER_UNIQ_SOLUTIONS); // randomize (if necessary) var-range-sequences on subsequent calls to solve the constraint
        if (p_vec= NEW_SOLN_VECTOR)
            {     // enough room in the soln-buffer exists -- get the solution-vector for the variables
            FOR_EACH_VAR(p_tab_elem, i)
            if (pval= get_var_value(p_tab_elem->name))
                p_vec[i]= *pval;
            END_FOR_EACH_VAR(p_tab_elem, i)

            ++soln_cnt;
            SET_BUF_SOLN_CNT(soln_cnt);
            }
        else    // too many solutions (or, zero variables in constraint) to store
            break;
        constraint= (order_type==ORDER_UNIQ_SOLUTIONS)? constraint:NULL;
        keep_prev_solns= (order_type==ORDER_UNIQ_SOLUTIONS);
        }

  doBufferSoln= TRUE;
```

```
 if (soln_cnt<= 2)
                return(soln_cnt> 0? soln_cnt+1: (stat> 0)? 1: 0);

 switch(order_type)     // -- order the solutions if so desired ----
                {
                case ORDER_LIKE_TOGETHER:     // try to gather like solutions together
                        qsort(solnBuf, soln_cnt, sizeof(Value)*varList_x, solution_cmp);
                        break;

                case ORDER_RANDOM:       // gather solutions in a random sequence - random
shuffle
                case ORDER_UNIQ_SOLUTIONS:
                        for(i= 0, half_cnt= soln_cnt/2; i< half_cnt; i++)
                                {
                                iy= half_cnt+ (random()%half_cnt);
                                SWAP_SOLUTIONS(i, iy);
                                }
                        break;

                case ORDER_DIFF_TOGETHER:     // try to gather "different" solutions
together - deterministic shuffle
                        if (soln_cnt== 3)
                                { // just swap the last two elements
                                SWAP_SOLUTIONS(1, 2);
                                return(soln_cnt> 0);
                                }
                        qsort(solnBuf, soln_cnt, sizeof(Value)*varList_x, solution_cmp);
                        for(i= 0, half_cnt= soln_cnt/2; i< half_cnt; i++) // intersperse the ordered
solutions
                                { // in preparation for shuffle, store the first half of the soln-vectors
in tmp-buffer
                                COPY_SOLN_to_TMPBUF(i);
                                }
                        for(i= 0, half_cnt= soln_cnt/2; i< half_cnt; i++) // intersperse the ordered
solutions
                                { // shuffle by mapping first half as: index-> 2*index, and the
second half as: index ->2*(index- half_cnt)+1.
                                MOVE_SOLUTION(2*i+1, i+ half_cnt);
                                COPY_SOLN_from_TMPBUF(2*i, i);
                                }
                                // after interspersing solutions, add some randomness to it too
                        for(i= 0, half_cnt= soln_cnt/2; i< half_cnt; i++)
                                {
                                if (brandom())
                                        {
```

```
                                        iy= half_cnt+ (random()%half_cnt);
                                        SWAP_SOLUTIONS(i, iy);
                                        }
                                }
                        break;

                default:
                        break;
                }

        return(soln_cnt> 0? soln_cnt+1: (stat> 0)? 1: 0);
 }
else
        {
 return(NEXT_BUF_SOLUTION != NULL);
        }
}

int CCONV SolveConstraintOrdered(char *constraint, int order_type)
{       // solve the given constraint (e.g. "X= Y+ 4, Y=2."); using a linear/interval solver as
needed
        // backtracks over the previous solution if the constraint is empty (i.e. NULL)
        // present the (nearly) ALL the solutions conforming to the given order (e.g.
ORDER_DIFF_TOGETHER)
        // returns, on first call (i.e. when constraint is non-NULL), (1+
the_total_count_of_solutions) (> 0) [false (=0)] if the constraint is [un]solvable;
        //              (note that in case of constraints without variables (e.g. "4= 4."),
total-no.-of-solutions is 0, though the constraint is provable.)
        // returns, on subsequent calls (i.e. when constraint is NULL), true (= 1) [false (=0)] if a
solution exists [does not exist];
        //      returns negative integer in error (e.g. if the constraint could not be parsed).
return(auxSolveConstraintOrdered(constraint, order_type, -1));
}

int CCONV SolveConstraintOrderedNSolns(char *constraint, int order_type, int max_soln)
{       // solve the given constraint (e.g. "X= Y+ 4, Y=2."); using a linear/interval solver as
needed
        // solve to find the given maximum no. (= max_soln) of solutions
        // backtracks over the previous solution if the constraint is empty (i.e. NULL)
        // present the solutions conforming to the given order (e.g. ORDER_DIFF_TOGETHER)
        // returns, on first call (i.e. when constraint is non-NULL), (1+
the_total_count_of_solutions) (> 0) [false (=0)] if the constraint is [un]solvable;
        //              (note that in case of constraints without variables (e.g. "4= 4."),
total-no.-of-solutions is 0, though the constraint is provable.)
        // returns, on subsequent calls (i.e. when constraint is NULL), true (= 1) [false (=0)] if a
```

```
solution exists [does not exist];
        //      returns negative integer in error (e.g. if the constraint could not be parsed).
return(auxSolveConstraintOrdered(constraint, order_type, max_soln));
}

int CCONV SolveConstraintLin(char *constraint)
{       // solve the given constraint (e.g. "X= Y+ 4, Y=2."); using a linear solver only
        // backtracks over the previous solution if the constraint is empty (i.e. NULL)
        // return true (=1) [false (=0)] if the constraint is [un]solvable;
        //      returns negative integer in error (e.g. if the constraint could not be parsed).

useIntervalSolver= FALSE;
return(auxSolveConstraint(constraint, FALSE, -1));
}

int CCONV SetPrecision(double precision)
{// set the precision for solving the constraint & for the solutions in the real domain
 // returns TRUE if ok
if (precision<= 0)
 return(FALSE);
Precision= precision;

return(TRUE);
}

int CCONV SetSolnDiffWt(int soln_diff_wt)
{// set the weight to indicate how "different" the solutions must be from each other in
Uniq_Soln_Order
        //      (the higher the weight, the more the solutions are "different".)
 // returns TRUE if ok
if (soln_diff_wt< 0)
 return(FALSE);
SolnDiffWt= soln_diff_wt;

return(TRUE);
}

int CCONV FractionalizeRational(int do_fractionalize)
{ // fractionalize all the rationals if do_fractionalize is TRUE; not otherwise (fractionalization
may slow things down a bit.)
fractionalizeRational= do_fractionalize;
return(TRUE);
}

int CCONV RoundoffFractionOnlyRational(int do_roundoff)
```

```
{ // roundoff fraction-only rationals (e.g. 2/3) if do_roundoff is TRUE; otherwise, return fraction-only rationals as fractions
roundoffFractionOnlyRational= do_roundoff;
return(TRUE);
}

int CCONV IsIndependentVar(char *var)
{       // return TRUE (1) if the given variable is independent (i.e. specified in an enumeration); FALSE (0) otherwise
TabElem *p_tab_elem;

if (var && (p_tab_elem= get_var(var)))
        return(p_tab_elem->is_independent_var);
else
        return(FALSE);
}

Value * CCONV GetValue(char *var)
{       // return the ptr to the value (in Value structure) of the given variable (e.g. "Area") if known;
        // return NULL on error (e.g. unknown varible, or variabe has no value, ...)
int i;
Value *val;
TabElem *p_tab_elem;

val= NULL;

if (!doBufferSoln)
  {
  val= get_var_value(var);
  }
else
  {     // retrieve appropriate value from the soln-buffer
  FOR_EACH_VAR(p_tab_elem, i)
        if (!strcmp(var, p_tab_elem->name))
          {     // found the var in varList
          val= CUR_BUF_SOLUTION(i);
          break;
          }
  END_FOR_EACH_VAR(p_tab_elem, i)
  }

return(val);
}
```

```
long CCONV GetValue_type(Value *val)    // return type (e.g. VAL_INTEGER) of the given
Value;
{
                                                        //      returns VAL_UNKNOWN in
error
return(val? val->type: VAL_UNKNOWN);
}

long CCONV GetVarValue_type(char *var) // return type (e.g. VAL_INTEGER) of the given
variable;
{
                                                        //      returns VAL_UNKNOWN in
error
Value *val;

return ((val= GetValue(var))? val->type: VAL_UNKNOWN);
}

long CCONV GetValue_int(Value *val)     // return integer value of the given Value structure;
{                                                       //      return
ERR_GETVALUE_INT in error (e.g. given structure is not integer)
return((val && val->type== VAL_INTEGER)? val->value.integer: ERR_GETVALUE_INT);
}

Rational CCONV GetValue_rational(Value *val)    // return rational value of the given Value
structure;
{                                   //      return <ERR_GETVALUE_RAT,
ERR_GETVALUE_INT, ERR_GETVALUE_INT> in error (e.g. given structure is not rational)
Rational rat;

if (val && val->type== VAL_RATIONAL_FLOAT || val && val->type==
VAL_RATIONAL_FRACTION)
 return(val->value.rational);
else
 {
 rat.real= ERR_GETVALUE_RAT;
 rat.num = rat.den = ERR_GETVALUE_INT;
 return(rat);
 }
}

double CCONV GetValue_rational_float(Value *val)        // return float rep. of the given
rational Value
{
return((val && (val->type== VAL_RATIONAL_FLOAT || val->type==
```

```
VAL_RATIONAL_FRACTION))? val->value.rational.real: ERR_GETVALUE_RAT);
}

long CCONV GetValue_rational_numer(Value *val)
{       // return numerator of the fractional rep. of the given rational Value
return((val && val->type== VAL_RATIONAL_FRACTION)? val->value.rational.num: 0);
}

long CCONV GetValue_rational_denom(Value *val)
{       // return denominator of the fractional rep. of the given rational Value
return((val && val->type== VAL_RATIONAL_FRACTION)? val->value.rational.den: 0);
}

Real CCONV GetValue_real(Value *val)    // return real value (i.e. lower & upper bound) from the given non-rational Value structure;
{                                       //      return <1, 0> in error (e.g. given structure is not real)
Real rl;

if (val && val->type== VAL_IRRATIONAL)
 return(val->value.real);
else
 {      // error
 rl.lower.val= 1;
 rl.upper.val= 0;
 rl.lower.is_infinite= rl.upper.is_infinite= TRUE;
 return(rl);
 }
}

double CCONV GetValue_real_lower(Value *val)  // return lower bound for the given non-rational real value
{                                             // return ERR_GETVALUE_REAL in error
return(val && val->type== VAL_IRRATIONAL?
        (val->value.real.lower.is_infinite? ERR_GETVALUE_REAL: val->value.real.lower.val): ERR_GETVALUE_REAL);
}

double CCONV GetValue_real_upper(Value *val)  // return upper bound for the given non-rational real value
{                                             // return ERR_GETVALUE_REAL in error
return(val && val->type== VAL_IRRATIONAL?
        (val->value.real.lower.is_infinite? ERR_GETVALUE_REAL: val->value.real.upper.val): ERR_GETVALUE_REAL);
}
```

```
BSTR CCONV VBGetValue_string(Value *val)     // VB wrapper for GetValue_string()
{
char *c_val;
BSTR vb_val;

if (!val)
        return (NULL);
vb_val= NULL;
if (c_val= GetValue_string(val))
        {
  vb_val= SysAllocStringByteLen(NULL, strlen(c_val)+1);          // alloc a new BSTR
        strcpy((char *)vb_val, c_val);
        }

return(vb_val);
}

BSTR CCONV VBGetVarValue(char *var) // VB wrapper for GetVarValue()
{
char *c_val;
BSTR vb_val;

if (!var)
        return (NULL);
vb_val= NULL;
if (c_val= GetVarValue(var))
        {
  vb_val= SysAllocStringByteLen(NULL, strlen(c_val)+1);          // alloc a new BSTR
        strcpy((char *)vb_val, c_val);
        }

return(vb_val);
}

char * CCONV GetVarValue(char *var)               // return (uniform) string representation of
the given variable
{                                                          //        returns NULL in error
Value *val;

if (!var)
        return (NULL);

return ((val= GetValue(var))? GetValue_string(val): NULL);
}
```

```
int CCONV GetVarValueBuf(char *var, int valuebuf_len, char valuebuf[])                    //
return (uniform) string representation of the given variable in the value-buffer
{                                                                       //        returns length (>0) of
the value in the valuebuf; returns <= 0 in error
Value *val;
char *val_str;
int val_len;

if (!var || !valuebuf || valuebuf_len<= 0)
        return (-1);
if (valuebuf_len> 0)
        valuebuf[0]= 0;
val_str= (val= GetValue(var))? GetValue_string(val): NULL;
val_len= val_str? strlen(val_str): 0;
if ((val_len> 0) && (valuebuf_len> val_len))
        {
        strcpy(valuebuf, val_str);
        free(val_str);
        }
else
        return (-1);

return (val_len);
}

char * CCONV GetValue_string(Value *val)            // return (uniform) string representation of
the given Value structure
{       // return NULL in error (e.g. given structure is not valid)
char str[1024], *p_tmp;
List *lst;

if (!val)
 return(NULL);
switch(val->type)
 {
 case VAL_INTEGER:
        sprintf(str, "%ld", val->value.integer);
        break;

 case VAL_RATIONAL_FLOAT:
        sprintf(str, "%f", val->value.rational.real);
        break;

 case VAL_RATIONAL_FRACTION:
        if (val->value.rational.den != 1)
```

```
            sprintf(str, "%d/%d", val->value.rational.num, val->value.rational.den);
          else
            sprintf(str, "%d", val->value.rational.num);
          break;

     case VAL_IRRATIONAL:
          if (val->value.real.lower.is_infinite)
            sprintf(str, "(<--, ");
          else
            sprintf(str, "(%f, ", val->value.real.lower.val);
          if (val->value.real.upper.is_infinite)
            sprintf(str+strlen(str), "-->)");
          else
            sprintf(str+strlen(str), "%f)", val->value.real.upper.val);
          break;

     case VAL_VAR:
          strcpy(str, "_");
          break;

     case VAL_STRING:
          strcpy(str, val->value.string);
          break;

     case VAL_FUNCTOR:
          sprintf(str, "%s/%d", val->value.functor.predicate, val->value.functor.arity);
          break;

     case VAL_LIST:
          strcpy(str, "[");
          for(lst= val->value.list; lst; lst= lst->next)
            {
            if (p_tmp= GetValue_string((Value *)lst->elem))
                        strcat(str, p_tmp);
            if (lst->next)
                        strcat(str, ", ");
            }
          strcat(str, "]");
          break;

     default:
                return(NULL);
     }

   return(strdup(str));
```

```
}

int CCONV IsFullyConstrained(char *constraint)
{ // returns TRUE if the given constraint is fully constrained (i.e. solvable & all variables are constant); FALSE otherwise (or in error)
 // (Check the immediately previous constraint if the given constraint is NULL.)
        // Note: It works only for the linear constraints because we use the linear-solver only here.
        //                  As such, it labels nonlinear constraints such as: Y^2= 2. as unconstrained.
        //                  The trouble with using interval-solver is that it may take a long time
        //                  to solve unconstrained equations e.g. X= Y+ 2. (because it finds all the solutions at once.)
int i;
Value *pval;
TabElem *p_tab_elem;

if (!constraint || SolveConstraintLin(constraint)> 0)
 {
 FOR_EACH_VAR(p_tab_elem, i)
        if (pval= doBufferSoln? CUR_BUF_SOLUTION(i): get_term_value(p_tab_elem->term))
         {          // ?? should we use get_var_value() here instead of get_term_value() ???
         if (pval->type== VAL_VAR ||
               (pval->type== VAL_IRRATIONAL &&
                     (pval->value.real.lower.is_infinite || pval->value.real.upper.is_infinite &&
                      ABS(pval->value.real.upper.val- pval->value.real.lower.val)> p_tab_elem->precision)))
               break;
         }
 END_FOR_EACH_VAR(p_tab_elem, i)
 return(i>= var_CNT);  // true if all var's are constant
 }
else
 return(FALSE);
}

BSTR CCONV VBPrintAllVarVals()         // VB wrapper (almost) for PrintAllVarVals()
{
char *c_val, buf[1024];
BSTR vb_val;

vb_val= NULL;
if (c_val= PrintAllVarVals(buf))
        {
 vb_val= SysAllocStringByteLen(NULL, strlen(c_val)+1);          // alloc a new BSTR
```

```
        strcpy((char *)vb_val, c_val);
        }

    return(vb_val);
    }

char * CCONV PrintAllVarVals(char buf[])
{ // Print all the var's with their values in the given buffer; return ptr to the given buffer
 // (assumes buffer is large enough to store all the var's.)
int i;
TabElem *p_tab_elem;
char tmp[256];

if (!buf)
  return(NULL);
buf[0]= 0;
FOR_EACH_VAR(p_tab_elem, i)
  sprintf(tmp, "%s: %s, ", p_tab_elem->name, GetValue_string(GetValue(p_tab_elem->name)));
  strcat(buf, tmp);
END_FOR_EACH_VAR(p_tab_elem, i)
if (strlen(buf)>= 2)
  buf[strlen(buf)-2]= 0;          // remove the last comma

return(buf);
}

char * CCONV PrintAllVarValsAllocate()
{ // Print all the var's with their values in an allocated buffer; return ptr to the given buffer
char buf[2048];

PrintAllVarVals(buf);

return(strdup(buf));
}

int CCONV Compile(char *p4_filename)     // compile & load the given p4_filename
(containing Prolog IV program)
{ // return true (1) if the file compiled ok, false (0) in error
int stat;

p4new_session();
p4make_call(P4MAKE_FUNC_1("compile", p4make_atom_from_cstring(p4_filename)));
stat= (p4next_solution()== P4SESSION_SOLUTION);
p4finish_session();
```

```
return(stat);
}

static int term_numden(P4TERM term, int *num, int *den)
{ // performs fractional representation for the given rational term i.e. rational= num/den (e.g. 4.5= 9/2)
 // uses numden/3 to achieve that. (note that you can call a Prolog predicate in the middle of another - in a recursive fashion.)
 // puts the numerator in the given num, and denominator in den.
 //  returns TRUE if succeeded in doing the transformation, FALSE otherwise.
int stat;
P4TERM tnum, tden;

if (!term)
        return(FALSE);
*num= *den= 0;

p4new_session(); // note that we start a session before making new terms - these new terms are valid only for this session
tnum= check_term(p4make_var());
tden= check_term(p4make_var());
p4make_call(P4MAKE_FUNC_3("numden", term, tnum, tden));
if ((stat= (p4next_solution()== P4SESSION_SOLUTION)) && p4is_integer(tnum) && p4is_integer(tden))
  {
  *num= p4val_lint(tnum);
  *den= p4val_lint(tden);
  }

p4finish_session();

return(stat);
}

static int numden(double rational, int *num, int *den)
{ // performs fractional representation for the given rational number i.e. rational= num/den (e.g. 4.5= 9/2)
 // uses numden/3 to achieve that. (note that you can call a Prolog predicate in the middle of another - in a recursive fashion.)
 // puts the numerator in the given num, and denominator in den.
 //  returns TRUE if succeeded in doing the transformation, FALSE otherwise.

return(term_numden(check_term(P4Make_Rational(rational)), num, den));
}
```

```
static int gcd(int x, int y)
{ // return the GCD of the two given integers
int num, den, tmp;

if (x< y)
  {     // swap x & y
  tmp= x;
  x= y;
  y= x;
  }
if (x== 0 || y== 0 || x==1 || y==1)
  return(1);
if (x==y || x==(-y))
  return(x);

if (numden(((double)x)/y, &num, &den))
  return((int)(x/num));
else
  return(1);
}



                // private functions
int GetInString(char buf[], int max_size)          // used in lexer - make it static later
{       // put input string (upto max_size) in the given buf[]; (usually called by YY_INPUT from
lexical analyzer)
        // return the no. of char's actually put in buf[]. (returns 0 if no char's put.)
int out_cnt;

out_cnt= MIN(max_size, (inExprBuf_cnt- inExprBuf_x));  // max. no. of char's to put in buf[]
memcpy(buf, inExprBuf+inExprBuf_x, out_cnt);
inExprBuf_x += out_cnt;

return(out_cnt);
}

static int bounds_of_real_var(P4TERM var, Real *realp)
{       // find the lower & upper bounds of the given (numeric) (real) var;
        //      return true (1) on success; false (0) on failure (e.g. given var is not numeric)
        //      (much of this func suggested by Pascal Bouvier of Prologianet)
        //      (we use glb/2 & lub/2 instead of bounds/3 so we can assign the
        //        infinite-bound to the appropriate end (i.e. lower/upper))
P4TERM tA, tB;
int done, status;
```

```
done= FALSE;
realp->lower.is_infinite= realp->upper.is_infinite= FALSE;
realp->lower.val= realp->upper.val= 0;

// well, for efficiency, let's use bounds/3 to check if the var is unbounded on either side
p4new_session();
tA = p4make_var();
tB = p4make_var();
p4make_call(P4MAKE_FUNC_3("bounds", var, tA, tB));
if ((status= p4next_solution())!= P4SESSION_SOLUTION)
        {
        if (status != P4SESSION_ERROR)
                realp->lower.is_infinite= realp->upper.is_infinite= TRUE; // <<-- KNJ: This is
not strictly correct, REVISIT it later -->>
        else
                {printf("\n***Encountered error 1 in
bounds_of_real_var()***\n");fflush(stdout);}
        done= TRUE;
        }
p4finish_session();
if (done)
        return(TRUE);

// first, check if the given var. is really unbounded on the lower side
p4new_session();
p4make_call(P4MAKE_FUNC_2("lt", var, P4Make_Rational(DEF_LOWER_BOUND)));
if ((status= p4next_solution())== P4SESSION_SOLUTION)
        {
        realp->lower.is_infinite= TRUE;
        }
else
        {       // not unbounded on the lower side - find the actual lower bound
        if (status== P4SESSION_ERROR)
                {printf("\n***Encountered error 2 in
bounds_of_real_var()***\n");fflush(stdout);}

        p4finish_session();
        p4new_session();
        tA = p4make_var();
        p4make_call(P4MAKE_FUNC_2("glb", var, tA));     // get the lower bound
        switch(p4next_solution())
                {
                case P4SESSION_END:             realp->lower.is_infinite= TRUE; break; // no
solution - the bound is infinite
                case P4SESSION_SOLUTION:        realp->lower.val= p4val_rational(tA);
```

```
                                                realp->lower.is_infinite=
SAME(realp->lower.val, DEF_LOWER_BOUND) ||
                                                (realp->lower.val<
DEF_LOWER_BOUND);
                                                break;
          case P4SESSION_ERROR:                 // error - given var may be
non-numeric
          default:      return(FALSE);
          }
     }
p4finish_session();

// next, check if the given var. is really unbounded on the upper side
p4new_session();
p4make_call(P4MAKE_FUNC_2("gt", var, P4Make_Rational(DEF_UPPER_BOUND)));
if ((status= p4next_solution())== P4SESSION_SOLUTION)
     {
     realp->upper.is_infinite= TRUE;
     }
else
     {    // not unbounded on the upper side - find the actual upper bound
     if (status== P4SESSION_ERROR)
          {printf("\n***Encountered error 3 in
bounds_of_real_var()***\n");fflush(stdout);}

     p4finish_session();
     p4new_session();
     tA = p4make_var();
     p4make_call(P4MAKE_FUNC_2("lub", var, tA));    // get the upper bound
     switch(p4next_solution())
          {
          case P4SESSION_END:           realp->upper.is_infinite= TRUE; break; // no
solution - the bound is infinite
          case P4SESSION_SOLUTION:      realp->upper.val= p4val_rational(tA);
                                        realp->upper.is_infinite=
SAME(realp->upper.val, DEF_UPPER_BOUND) ||
                                                (realp->lower.val>
DEF_UPPER_BOUND);
                                        break;
          case P4SESSION_ERROR:                 // error - given var may be
non-numeric
          default:      return(FALSE);
          }
```

```
        }
p4finish_session();

return(TRUE);
}

static P4TERM get_value_term(Value *val)
{ // return the Prolog IV term for the given basic value (in Value struct)
if (!val)
  return(NULL);
switch(val->type)
  {
  case VAL_INTEGER: return(check_term(p4make_lint(val->value.integer))); break;
        case VAL_RATIONAL_FLOAT:
return(check_term(P4Make_Rational(val->value.rational.real))); break;
  case VAL_RATIONAL_FRACTION:
        return(((val->value.rational.den != 1) && (val->value.rational.den != 0))?

check_term(P4Make_Rational((double)val->value.rational.num/(double)val->value.rational.den)
):
              check_term(P4Make_Rational((double)val->value.rational.num)));
              break;
        case VAL_IRRATIONAL:    // ?? what is the P4-call to make a P4term for an irrational
???
              return(NULL);        break;
  case VAL_VAR: return(NULL); break;
  case VAL_STRING:
return(check_term(p4make_atom_from_cstring(val->value.string)));        break;
        case VAL_FUNCTOR:        // ?? what is the P4-call to make a P4term for a functor ???
              return(NULL);        break;
  case VAL_LIST: return(check_term(make_valslist2term(val->value.list))); break;

  default:       return(NULL);
  }

return(NULL);
}

static Value * get_var_value(char *var)
{ // return the (ptr to) value (in Value struct) of the given PrologIV variable
  // (It finds the basic value through get_term_value(), and
  //      then interprets that further for the given var & its type (e.g. precision/type considerations)
)
TabElem *p_tab_elem;
Value *val;
```

```
if (!var)
        return(NULL);

if (!(p_tab_elem= get_var(var)) || !(val= get_term_value((P4TERM)p_tab_elem->term)))
        {
        if (!val)
                {printf("\n***Recvd NULL value for the given variable %s;***\n",
var);fflush(stdout);}
 return(NULL);
        }
switch(val->type)
 {
 case VAL_VAR:
                /*      <<-- KNJ: ignore all this for now - for efficiency's sake -- REVISIT it later
-->>
        if (useIntervalSolver && bounds_of_real_var((P4TERM)p_tab_elem->term,
&val->value.real))
         {
         val->type= VAL_IRRATIONAL;
         if (!val->value.real.lower.is_infinite &&
                !val->value.real.upper.is_infinite &&
                ABS(val->value.real.upper.val- val->value.real.lower.val)<=
p_tab_elem->precision)
                        { // the range is smaller than the variable's precision - it can be represented
as a rational
                        val->type= VAL_RATIONAL_FLOAT;
                        val->value.rational.real= (val->value.real.lower.val+
val->value.real.upper.val)/2;
                        val->value.rational.num= val->value.rational.den= 0;
                        if (p_tab_elem->type== VAL_RATIONAL_FRACTION)
                                {
                                numden(val->value.rational.real, &val->value.rational.num,
&val->value.rational.den);
                                val->type= VAL_RATIONAL_FRACTION;
                                }
                        }
         }
                */
        break;

 case VAL_IRRATIONAL:
#ifdef OLD_IRRATIONAL
        if (!val->value.real.lower.is_infinite &&
                        !val->value.real.upper.is_infinite &&
                        ABS(val->value.real.upper.val- val->value.real.lower.val)<=
```

```
p_tab_elem->precision)
            { // the range is smaller than the variable's precision - it can be represented as a rational
                        // this introduces errors (due to roundoff) which, however small, are
unacceptable for ETS work
            val->type= VAL_RATIONAL_FLOAT;
            val->value.rational.real= align_val_with_precision((val->value.real.lower.val+
val->value.real.upper.val)/2, p_tab_elem->precision);
            val->value.rational.num= val->value.rational.den= 0;
            if (p_tab_elem->type== VAL_RATIONAL_FRACTION)
                        {
                        numden(val->value.rational.real, &val->value.rational.num,
&val->value.rational.den);
                        val->type= VAL_RATIONAL_FRACTION;
                        }
            }
#endif /* OLD_IRRATIONAL */
        break;

 case VAL_RATIONAL_FLOAT:
        val->value.rational.real= align_val_with_precision(val->value.rational.real,
p_tab_elem->precision);
        if (p_tab_elem->type== VAL_RATIONAL_FRACTION)
            {
            numden(val->value.rational.real, &val->value.rational.num, &val->value.rational.den);
            val->type= VAL_RATIONAL_FRACTION;
            }
        break;

 default:
        break;
 }

return(val);
}

static int are_equal_terms(P4TERM term1, P4TERM term2)
{       // returns true if the two given terms are equal
int stat;

if (!term1 || !term2)
        return(FALSE);

p4new_session();
p4make_call(P4EQ(term1, term2));
stat= (p4next_solution()== P4SESSION_SOLUTION);
```

```
p4finish_session();

return(stat);
}

static Value *get_term_value(P4TERM term)
{ // return the basic value (in Value struct) of the given Prolog IV term
Value *val, *p_tmp_val;
int term_type;

if (!term  || p4is_nil(term))
  return(NULL);

//val= (Value *)calloc(1, sizeof(Value));
val= NEW_VALUE;
switch(term_type= P4WHAT_IS(term))
  {
  case P4INTEGER:
        val->type= VAL_INTEGER;
        val->value.integer= p4val_lint(term);
        break;

  case P4FLOAT:
        val->type= VAL_RATIONAL_FLOAT;
        val->value.rational.real= p4val_double(term);
        val->value.rational.num= val->value.rational.den= 0;
        break;

  case P4RATIONAL:
        val->type= VAL_RATIONAL_FLOAT;
        val->value.rational.real= p4val_rational(term);
        if (!roundoffFractionOnlyRational && !are_equal_terms(term,
check_term(P4Make_Rational(val->value.rational.real))))
                {       // if the rational cannot be represented in non-fractional terms
                val->type= VAL_RATIONAL_FRACTION;
                term_numden(term, &val->value.rational.num, &val->value.rational.den);
                }
        break;

  case P4ATOM:
        val->type= VAL_STRING;
        val->value.string= p4val_cstring(term);      // do we need to realloc the string ?
        break;

  case P4DOT:
```

```
        val->type= VAL_LIST;
        p_tmp_val= get_term_value(p4cdr(term));
        val->value.list= cons(get_term_value(p4car(term)), p_tmp_val? p_tmp_val->value.list:
NULL);
        break;

  case P4FUNCTOR:
        val->type= VAL_FUNCTOR;
        val->value.functor.predicate= p4val_cstring(term);
        val->value.functor.arity= p4get_arity(term);
        break;

  case P4VAR:
        val->type= VAL_VAR;
        if (bounds_of_real_var(term, &val->value.real) &&  // <<-- added check: KNJ
              !val->value.real.lower.is_infinite && !val->value.real.upper.is_infinite)
          val->type= VAL_IRRATIONAL;
        break;

  default:
              printf("\n***Recvd unknown value-type for the given variable;***\n");
fflush(stdout);
        //free(val);
              return(NULL);
  }

if (p4errno!= 0)
              {printf("\n***Unknown error occurred when getting value of a
variable;***\n");fflush(stdout);}

return(val);
}

              // aux functions
static int hash(char *str)
{
int i, h;

for(i= h= 0; str[i]; i++) h += str[i]*i;

return(h% VAR_TABLE_SIZE);
}

static TabElem *get_var(char *var)
{    // ptr to the appropriate variable-entry in the var-table
```

```
TabElem *p;

for(p= varTable[hash(var)]; p && p->name && strncmp(p->name, var,
MAX_VAR_NAME_LEN); p= p->next);
return(p);
}

static P4TERM get_var_term(char *var)
{   // return the term associated with the variable
TabElem *p;

p= get_var(var);
return(p? (P4TERM)p->term: NULL);
}

static TabElem *get_term_tabelem(P4TERM term)
{   // return the var-table-element associated with the given term
int i;
TabElem *p_tab_elem;

FOR_EACH_VAR(p_tab_elem, i)
  if (p_tab_elem->term== term)
              return(p_tab_elem);
END_FOR_EACH_VAR(p_tab_elem, i)

return(NULL);
}

static int set_term_on_grid(P4TERM term)
{
TabElem *p_tab_elem;
if (p_tab_elem= get_term_tabelem(term))
        p_tab_elem->ongrid= TRUE;

return TRUE;
}

static int reset_term_on_grid(P4TERM term)
{
TabElem *p_tab_elem;
if (p_tab_elem= get_term_tabelem(term))
        p_tab_elem->ongrid= FALSE;

return TRUE;
}
```

```
static void insert_const(P4TERM term, Value *const_value)
{	// insert the given {term, constant value} pair in constBuf[]
ADD_constBuf(term, *const_value);
return;
}

static Value *get_const_value(P4TERM term)
{	// return the Value associated with the given term if the term is numeric-constant; NULL
otherwise.
int i;

for(i= 0; (i< constBuf_x) && (constBuf[i].term != term); i++);
return((i< constBuf_x)? &constBuf[i].value: NULL);
}

static P4TERM insert_var(char *var)
{ // insert given var (with an associated Prolog term) to the var-table if not already present
	//	return ptr to the term associated with the var.
TabElem *p;
int h;

if (!strcmp(var, "_"))
 {	// anonymous variable - just return a Term for it
 return(check_term(p4make_var()));
 }
else if (!(p= get_var(var)))
 {	// non-anonymous variable - store in variable table
 p= NEW_TAB_ELEM;
	// initialize variable to default type
 p->type= DEF_VAR_TYPE; // <-- actual type (e.g. int, ...) may be put later
	p->is_independent_var = FALSE;
 p->precision= DEF_PRECISION;
 p->ongrid= DEF_GRID;
 strncpy(p->name, var, MAX_VAR_NAME_LEN);
 p->term= check_term(p4make_var());
 p->next= varTable[h= hash(var)];
 varTable[h]= p;
 ADD_varList(p);
 }
return((P4TERM)p->term);
}

static P4TERM insert_var_with_precision(char *var, double precision)
{ // insert (or modify its attribs if preexistent) given var (with an associated Prolog term) to the
var-table;
```

```
		//	(attach the given precision to the variable)
		//	return ptr to the term associated with the var.
	TabElem *p;
	int h;

	if (!(p= get_var(var)))
	 {
	 p= NEW_TAB_ELEM;
		// initialize variable to default type
	 p->type= DEF_VAR_TYPE; // <-- actual type (e.g. int, ...) may be put later
		p->is_independent_var = FALSE;
	 p->ongrid= DEF_GRID;
	 strncpy(p->name, var, MAX_VAR_NAME_LEN);
	 p->term= check_term(p4make_var());
	 p->next= varTable[h= hash(var)];
	 varTable[h]= p;
	 ADD_varList(p);
	 }

	p->precision= precision;

	return((P4TERM)p->term);
	}

	static int mark_var_type(char *var, int type)
	{	// mark the type of the given variable; return TRUE if ok, FALSE in error
	TabElem *p;

	if (!(p= get_var(var)))
	 return(FALSE);
	p->type= type;

	return(TRUE);
	}

	static int mark_term_var_type(P4TERM term, int type)
	// mark the type of the given term (in the var-table) as given;
	//	returns TRUE if the action done successfully; FALSE otherwise.
	{
	TabElem *p_tab_elem;

	if (p_tab_elem= get_term_tabelem(term))
		{
		p_tab_elem->type= type;
		return(TRUE);
```

```
        }
else
        return(FALSE);
}

static int mark_term_list_type(List *var_lst, int type)
{       // mark the type of the variables in the given variable-term-list; return TRUE if ok,
FALSE in error
for( ; var_lst; var_lst= var_lst->next)
  mark_term_var_type((char *)var_lst->elem, type);

return(TRUE);
}

static List *cons(void *elem, List *lst)
{    // cons the given elem to the [front of the] list
List *p;

if (!elem)
  return(lst);
if (p= calloc(1, sizeof(List)))
  {
  p->elem= elem;
  p->next= lst;
  p->elem_cnt= lst? lst->elem_cnt+1: 1;
  }
else
        {
        printf("\n***Unable to calloc List in cons()***\n");fflush(stdout);
  return(NULL);
        }

return(p);
}

static List *ncons(List *lst, void *elem)
{    // ncons the given elem to the [end of the] list
List *p, *l;

if (!elem)
  return(lst);
if (p= calloc(1, sizeof(List)))
  {
  p->elem= elem;
```

```
 p->next= NULL;
 p->elem_cnt= 1;
 }
else
 {
      printf("\n***Unable to calloc List in ncons()***\n");fflush(stdout);
 return(NULL);
 }
if (lst)
 {
 lst->elem_cnt++;
 for(l= lst; l->next; l= l->next);
 l->next= p;
 }
else
 lst= p;

return(lst);
}



      // --- action-routines ---

static int find_setof_soln(P4TERM goal)
{ // find (& buffer & reorder) the set of solns for the given goal;
 // return the Prolog-returned status of the call
 // Note that the P4 must have been initialized with the following query once before calling
setof/3 or bagof/3:
 //      def_array(tab_bagof, 100), record(block_limit, 0), record(last_bag_bound, 0).
int i, soln_cnt, status;
P4TERM var_set, R, vec;
Value *p_vec;

if (!doBufferSoln)
 return(FALSE);

for(var_set= check_term(p4make_nil()), i= varList_x-1; i>= 0; i--)   // build a term for the set of
var's
 var_set= check_term(p4make_dot((P4TERM) varList[i]->term, var_set));
R= check_term(p4make_var());
goal= P4MAKE_FUNC_3("setof", var_set, goal, R);
P4MAKE_CALL(goal);
if ((status= (p4next_solution()== P4SESSION_SOLUTION)) && !p4is_nil(R))
 {      // found the solns - buffer them
```

```
  for(soln_cnt= 0; !p4is_nil(R); R= p4cdr(R))
        {
        if (p_vec= NEW_SOLN_VECTOR)
         {      // enough room in the soln-buffer exists
         for(vec= p4car(R), i= 0; !p4is_nil(vec); vec= p4cdr(vec), i++)
                {
                p_vec[i]= *get_term_value(p4car(vec));
                }
         soln_cnt++;
         }
        }
  SET_BUF_SOLN_CNT(soln_cnt);
        // **** how about ordering the solutions ----- in next pass
  }

return(status);
}

static P4TERM split_var_range_term()
{ // return the split-var-range term (e.g. intsplit([X, Y]), realsplit([Z])) for the var's in the current
constraint
 // (useful only while solving with the interval solver.)
        // (we set the default var-type to be real.)
        // NOTE: if the variable is UNBOUND in the clause, a split (intsplit/realsplit)
        //                                 may give rise to VB overflow !!
int i;
P4TERM lst_term, term, intsplit, realsplit;
//P4TERM anon;
TabElem *p_tab_elem;

if (!useIntervalSolver)                  // can we use intsplit/realsplit outside of the interval-solver
? NO
 return(NULL);

intsplit= realsplit= NULL;
FOR_EACH_VAR(p_tab_elem, i)
 switch(p_tab_elem->type)
        {
        case VAL_INTEGER:
         lst_term= check_term(p4make_dot(p_tab_elem->term, p4make_nil()));    // X -> [X]
         term= P4MAKE_FUNC_3("intsplit", lst_term,
check_term(p4make_atom_from_cstring("smallest_domain")),
check_term(P4Make_Rational(p_tab_elem->precision)));
         intsplit= P4AND(intsplit, term);
         break;
```

```
        case VAL_RATIONAL_FLOAT:
        case VAL_RATIONAL_FRACTION:
        case VAL_IRRATIONAL:
        case VAL_REAL:
         lst_term= check_term(p4make_dot(p_tab_elem->term, p4make_nil()));   // X -> [X]
         term= P4MAKE_FUNC_3("realsplit", lst_term,
check_term(p4make_atom_from_cstring("smallest_domain")),
check_term(P4Make_Rational(p_tab_elem->precision)));
         realsplit= P4AND(realsplit, term);
         break;

        case VAL_LIST:
        case VAL_SYMBOL:
         break;

        default:
         break;
        }
END_FOR_EACH_VAR(p_tab_elem, i)
              // <<--- how does it work in case the anon var. is nonnumber e.g. list ??
        // NOTE: if the variable is UNBOUND in the clause, a split (intsplit/realsplit) may give
rise to VB overflow !!
#ifdef NOIGNORE_ANON_SPLIT          // don't do it until really needed
FOR_EACH_ANON_VAR(anon)
 lst_term= p4make_dot(anon, p4make_nil());        // X -> [X]
 term= P4MAKE_FUNC_3("realsplit", lst_term,
p4make_atom_from_cstring("smallest_domain"));
 realsplit= realsplit? P4AND(realsplit, term): term;
END_FOR_EACH_ANON_VAR(anon)
#endif /* NOIGNORE_ANON_SPLIT */

return(P4AND(intsplit, realsplit));
}

static P4TERM set_var_bounds_term()
{ // return the term setting bounds (e.g. -BOUND < X < BOUND) for all the variables used in the
constraint
 // (useful only for the interval solver.)
 // (ideally, we should first check to see if the var has a bound already set in the constraint.
 //      only when no such bound is set should we set a default one.
int i;
P4TERM term, bound;
//P4TERM anon;
TabElem *p_tab_elem;
```

```
if (!useIntervalSolver)
  return(NULL);

term= NULL;
FOR_EACH_VAR(p_tab_elem, i)
if (p_tab_elem->type== VAL_INTEGER || p_tab_elem->type== VAL_REAL ||
                p_tab_elem->type== VAL_RATIONAL_FLOAT || p_tab_elem->type==
VAL_RATIONAL_FRACTION)
        {
        bound= P4MAKE_FUNC_3("cc", p_tab_elem->term,
check_term(P4Make_Rational(DEF_LOWER_BOUND)),
check_term(P4Make_Rational(DEF_UPPER_BOUND)));
        term= P4AND(term, bound);
        }
END_FOR_EACH_VAR(p_tab_elem, i)

// <<--- KNJ: try removing the constraints on anonymous var's        <<----
//FOR_EACH_ANON_VAR(anon)
//  bound= P4MAKE_FUNC_3("cc", anon, P4Make_Rational(DEF_LOWER_BOUND),
P4Make_Rational(DEF_UPPER_BOUND));
//  term= P4AND(term, bound);
//END_FOR_EACH_ANON_VAR(anon)
// <<--- KNJ: try removing the constraints on anonymous var's        <<----

return(term);
}

static P4TERM vars_multiple_of_precision_term()
{       // return a term binding variables to be integer-multiples of their precision
        //        (e.g. int(_N), X= _N* X_precision)
        // NOTE: This constraint excludes irrational variables (e.g. PI)!
int i;
P4TERM term, t;
TabElem *p_tab_elem;

term= NULL;
FOR_EACH_VAR(p_tab_elem, i)
if (p_tab_elem->ongrid && (p_tab_elem->type== VAL_REAL || p_tab_elem->type==
VAL_RATIONAL_FLOAT || p_tab_elem->type== VAL_RATIONAL_FRACTION))
        {       // for each variable, add: var_with_precision(Var, Precision).
        t= P4MAKE_FUNC_2("var_with_precision", p_tab_elem->term,
check_term(P4Make_Rational(p_tab_elem->precision)));
        term= P4AND(term, t);
        }
END_FOR_EACH_VAR(p_tab_elem, i)
```

```
return(term);
}

static P4TERM buffered_func_terms()
{ // return the term containing the buffered functions (e.g. X= mean(X,Y) -> X= _R,
mean(_R,X,Y), where mean(_R,X,Y) is buffered)
return(combine_terms_array(funcTermBuf, funcTermBuf_x, TRUE, FALSE, NULL, FALSE));
}

static P4TERM get_exclude_solutions_term(int randomize)
{       // return a term corresponding to the negation of all the previous solutions of independent
var's e.g. (X=/= 4, Y=/=5, X=/= 10, Y=/=15).
TabElem *p_tab_elem;
P4TERM t, term;
Value *p_soln_vec;
int x_var, x_soln_vec;
int i;
char exclude_soln;

term= NULL;
exclude_soln= TRUE;
FOR_EACH_VAR(p_tab_elem, x_var)
        if (p_tab_elem->is_independent_var)
                {       // exclude only the variables to be used to generate uniq solns (e.q.
enumerate variables)
                FOR_EACH_SOLN_VECTOR(p_soln_vec, x_soln_vec)
                        if (randomize) // randomize the exclusion of solutions
                                for(i=0, exclude_soln= FALSE; !exclude_soln && (i<
SolnDiffWt); i++)
                                        exclude_soln= brandom();
                        if (exclude_soln && (t= get_value_term(&p_soln_vec[x_var])))
                                {
                                t= P4NEQ(p_tab_elem->term, t);
                                term= P4AND(term, t);
                                }
                END_FOR_EACH_SOLN_VECTOR(p_soln_vec, x_soln_vec)
                }
END_FOR_EACH_VAR(p_tab_elem, x_var)

return(term);
}

static P4TERM get_enum_ranges_term(int randomize)
{       // return a term corresponding to the enumerated-ranges (which were stored in the
enumRange buffer)
```

```
return(enumRangeTermBuf_CNT<=0? NULL: combine_terms_array(enumRangeTermBuf,
enumRangeTermBuf_CNT, TRUE, randomize, NULL, FALSE));
}

static P4TERM get_var_types_term(int randomize)
{       // return a term corresponding to the combined var-types (e.g. int(X); neq_vars(X,Y))
(which were stored in the VAR_TYPES buffer)

return(varTypesTermBuf_CNT<=0? NULL: combine_terms_array(varTypesTermBuf,
varTypesTermBuf_CNT, TRUE, randomize, NULL, FALSE));
}

static P4TERM pregoal_addenda_terms()
{       // return terms (e.g. -BOUND< X < BOUND) to be added before the current goal
P4TERM term, types_term, bound_term, prec_term;
P4TERM exclude_solns_term, var_interval_term;

term= types_term= get_var_types_term(FALSE);
bound_term= NULL; //        set_var_bounds_term();        <<-- KNJ: remove it for now <<--
term= P4AND(term, bound_term);
exclude_solns_term= RandomizeConstraints? get_exclude_solutions_term(TRUE): NULL;
term= P4AND(term, exclude_solns_term);
prec_term= vars_multiple_of_precision_term();
term= P4AND(term, prec_term);
var_interval_term= get_enum_ranges_term(RandomizeConstraints);
term= P4AND(term, var_interval_term);

return(term);
}

static P4TERM postgoal_addenda_terms()
{       // return terms (e.g. intsplit/1, func-terms) to be added after the current goal
P4TERM term, func_term;
//P4TERM split_term;

term= func_term= buffered_func_terms();
// split_term= split_var_range_term();        // No need for splitting vars - and it may cause some
overflow problems in VB
// term= P4AND(term, split_term);

return(term);
}

static P4TERM append_func_defs(P4TERM term)
{ // append (before/after the term ?) the function def's to the given term
```

```
        // returns the appended term
P4TERM t, func_term;

func_term= buffered_func_terms();
t= P4AND(term, func_term);              // for now, append the func_term AFTER the given term
INIT_funcTermBuf;          // function-terms already consumed here - reset the functerms-buffer

return(t);
}

static P4TERM insert_func_defs(P4TERM var_defs, P4TERM expr_lst)
{ // insert the function def's between the var_defs and expr_lst
        // returns P4AND(append_func_defs(var_defs), expr_lst)
P4TERM term;

term= append_func_defs(var_defs);
term= P4AND(term, expr_lst);

return(term);
}

static BOOLEAN process_call_to_prolog(P4TERM goal)
{ // call Prolog IV with the given goal; return the result (i.e. TRUE/FALSE) of the call
 // sets the BOOLEAN var resultFromProlog4 to the result.
P4TERM pregoal, postgoal;

resultFromProlog4= 0;
if (semError!= 0)
 return(FALSE);

if (!goal)
 return(FALSE);

if (pregoal= pregoal_addenda_terms())          // terms to add before goal
        goal = P4AND(pregoal, goal);
if (postgoal= postgoal_addenda_terms())        // terms to add after goal
        goal = P4AND(goal, postgoal);

/*      -- P4 setof/3 & bagof/3 are buggy - hence we simulate them ourselves
if (doBufferSoln)
 {      // call setof/3 - buffer (& order) all the solutions, return the result of call
 return(resultFromProlog4= find_setof_soln(goal));
 }
```

```
else
*/

P4MAKE_CALL(goal);
resultFromProlog4= p4next_solution();

/* -- Prints message incorrectly on time-limit-forced abortion
if (resultFromProlog4==P4SESSION_ERROR)
        {printf("\n***Encountered error in process_call_to_prolog(), errno: %d***\n",
p4errno);fflush(stdout);}
*/

return(resultFromProlog4== P4SESSION_SOLUTION);
}

static P4TERM combine_terms_array(P4TERM terms[], long size_terms, int do_conjunct, int
randomize, P4TERM var, int var_eq)
{// returns a combined terms for the elements (or, if var is non-NULL: var= Element (if var_eq is
TRUE) (or, if var_eq is FALSE, var =/= Element)) from the given array of terms :
        //              use conjunct to combine terms if do_conjunct is TRUE, disjunct
otherwise;
        // (randomize the array before combining if randomize is TRUE.)
        // (note that it shuffles the terms, when randomizing, in the given terms-buffer itself)
int i;
P4TERM t, cmbnd_term;
#define MAX_SHUFFLE             (51)

if (size_terms<= 0)
 return(NULL);
if (size_terms==1)
 return(terms[0]);

if (randomize)
        {
        int x, y, max_shuffle;
        P4TERM tmp;
                        // graduated scale for max-shuffling
        max_shuffle= (size_terms<= 10)? 3* size_terms:
                                                (size_terms<= 20)? 2* size_terms+
10:
                                                MAX_SHUFFLE;
        for(i= 0; i< max_shuffle; i++)
                {       // shuffle the terms in the terms-buffer
                x= random()% size_terms;
                y= random()% size_terms;
```

```
                tmp= terms[x];
                terms[x]= terms[y];
                terms[y]= tmp;
                }
        }
for(cmbnd_term= NULL, i= 0; i< size_terms; i++)
        {
        t= !var? terms[i]: var_eq? P4EQ(var, terms[i]): P4NEQ(var, terms[i]);
        cmbnd_term= do_conjunct? P4AND(cmbnd_term, t): P4OR(cmbnd_term, t);
        }

return(cmbnd_term);
}

static P4TERM combine_terms_list(List *terms_lst, int do_conjunct, int randomize, P4TERM
var, int var_eq)
{// returns a combined terms for the elements (or, if var is non-NULL: var= Element (if var_eq is
TRUE) (or, if var_eq is FALSE, var =/= Element)) from the given list of terms :
        //              use conjunct to combine terms if do_conjunct is TRUE, disjunct
otherwise;
        // (randomize the array before combining if randomize is TRUE.)
P4TERM t, cmbnd_term;

if (!randomize)
        {
        for(cmbnd_term= NULL; terms_lst; terms_lst= terms_lst->next)
                {
                t= !var? terms_lst->elem: var_eq? P4EQ(var, terms_lst->elem): P4NEQ(var,
terms_lst->elem);
                cmbnd_term= do_conjunct? P4AND(cmbnd_term, t): P4OR(cmbnd_term, t);
                }
        }
else
        {       // put the list-terms in an array, call combine_terms_array() with the array
        P4TERM *terms;
        List *lst;
        int i;

        for(i= 0, lst= terms_lst; lst; lst= lst->next, i++);
        if (!(terms= calloc(i, sizeof(P4TERM))))
                {
                printf("\n***Unable to calloc P4TERM-array in
combine_terms_list()***\n");fflush(stdout);
                return(NULL);           // return NULL on error
                }
```

```
	for(i= 0, lst= terms_lst; lst; lst= lst->next, i++)
		terms[i]= lst->elem;
	cmbnd_term= combine_terms_array(terms, i, do_conjunct, randomize, var, var_eq);
	free(terms);
	}
return(cmbnd_term);
}

static char *map_func_name(char *given_func_name)
		// check if the given function-name needs to be mapped e.g. gcd -> gcdtemp
		// return the properly mapped function name
{
int i;

for(i= 0; FuncRenameList[i].func_name && strcmp(given_func_name,
FuncRenameList[i].func_name); i++);
return(FuncRenameList[i].func_name? FuncRenameList[i].map_to_name: given_func_name);
}

static P4TERM unop_term(int op, P4TERM term)
{ // make unary op term for standard op's, return the result term (e.g. "- X"  =>  "uminus(Res,
X)", return Res).
 // (Note: This function can handle only the PrologIV-standard unary operators; use
make_functor() for nonstandard operators (e.g. abs, factorial) )
char *func;
P4TERM res;

if (!term)
 {
 semError= ERR_NULL_TERM;
 return(NULL);
 }

switch(op)
 {  // at some point, we need to decide between approx. & exact solvers (e.g. uminuslin vs
uminus)
 case '-':  func= useIntervalSolver? ".-.": "-"; break;
 case '+':  func= useIntervalSolver? ".+.": "+";  break;
 default:   return(NULL);
 }
func= map_func_name(func);
if (!(res= P4MAKE_FUNC_1(func, term)))
 {
 semError= ERR_MAKING_FUNCTOR;
 return(NULL);
```

```
  }
 return(res);
 }

 static P4TERM binop_arith_term(P4TERM term1, int op, P4TERM term2)
 {  // make binary op term for standard arithmetic op's, return the result term (e.g. " X + Y"  =>
 "plus(Res, X, Y)".
        // Returns (ptr to the term representing) the result of the computation rather than the
 Boolean status of the call.
        // Note that these are called using arity 3 Prolog IV predicates.
  // (Note: This function can handle only the PrologIV-standard binary operators; use
 make_functor() for nonstandard operators (e.g. mod, div) )
 char *func;
 P4TERM res;

 if (!term1 || !term2)
  {
  semError= ERR_NULL_TERM;
  return(NULL);
  }

 switch(op)
  {   // at some point, we need to decide between approx. & exact solvers (e.g. minuslin vs minus)
  case '+':  func= useIntervalSolver? ".+.": "+";  break;
  case '-':  func= useIntervalSolver? ".-.": "-";  break;
  case '*':  func= useIntervalSolver? ".*.": "*";  break;
  case '/':  func= useIntervalSolver? "./.": "/";  break;
  case '%':  func= "mod";  break;
  case '\\': func= "intdiv";  break;
  case '^':  func= ".^.";  break; // power/2
  // case '~':  func= "~";  break;
  default:         return(NULL);
  }
 func= map_func_name(func);
 if (!(res= P4MAKE_FUNC_2(func, term1, term2)))
  {
  semError= ERR_MAKING_FUNCTOR;
  return(NULL);
  }
 return(res);
 }

 static P4TERM binop_bool_term(P4TERM term1, int op, P4TERM term2)
 {  // make binary op term for standard boolean op's (e.g. " X, Y"  => "and(X, Y)".
 char *func;
```

```
P4TERM res;

if (!term1 || !term2)
 {
 semError= ERR_NULL_TERM;
 return(NULL);
 }

switch(op)
 {  // at some point, we need to decide between approx. & exact solvers (e.g. minuslin vs minus)
 case EQ:   func= "=";  break;
 case NEQ:  func= "dif";  break;
 case ',':  func= ",";  break;              // note that ,/2 is different from and/2 in Prolog IV
 case ';':  func= ";";  break;              // note that ;/2 is different from or/2 in Prolog IV
 case LT:  func= useIntervalSolver? "lt": "ltlin";  break;
 case LE:  func= useIntervalSolver? "le": "lelin";  break;
 case GT:  func= useIntervalSolver? "gt": "gtlin";  break;
 case GE:  func= useIntervalSolver? "ge": "gelin";  break;
 default:        return(NULL);
 }
func= map_func_name(func);
if (!(res= P4MAKE_FUNC_2(func, term1, term2)))
 {
 semError= ERR_MAKING_FUNCTOR;
 return(NULL);
 }
return(res);
}

static P4TERM make_termlist2term(List *lst)
{ // convert the given list of P4-terms to Prolog IV list-term
        // recursive implementation to maintain the order of elem's
                        // free the given list ??
return(lst? check_term(p4make_dot((P4TERM) lst->elem, make_termlist2term(lst->next))):
check_term(p4make_nil()));
}

static P4TERM make_valslist2term(List *lst)
{ // convert the given list of Values to Prolog IV list-term
        // recursive implementation to maintain the order of elem's
return(lst? check_term(p4make_dot(get_value_term(lst->elem), make_valslist2term(lst->next))):
check_term(p4make_nil()));
}

static P4TERM make_int_rel(List *lst)
```

```
{ // make (& return) a conjunct of int/1 terms (e.g. int(X), int(Y)) for the given list
P4TERM t, term;

for(term= NULL; lst; lst= lst->next)
        {
        t= P4MAKE_FUNC_1("int", (P4TERM) lst->elem);
 term= P4AND(term, t);
        }

// add the actual int_rel-term to the VAR_TYPES buffer -    so we can later manipulate it
if (term)
        ADD_VAR_TYPES_TERM(term);

//      return just a placeholder since we already added the int_rel-term to the buffer
return(P4TRUE);
}

static P4TERM make_real_rel(List *lst)
{ // make (& return) a conjunct of real/1 terms (e.g. real(X), real(Y)) for the given list
P4TERM t, term;

for(term= NULL; lst; lst= lst->next)
        {
        t= P4MAKE_FUNC_1("real", (P4TERM) lst->elem);
 term= P4AND(term, t);
        }

// add the actual real_rel-term to the VAR_TYPES buffer -  so we can later manipulate it
if (term)
        ADD_VAR_TYPES_TERM(term);

//      return just a placeholder since we already added the real_rel-term to the buffer
return(P4TRUE);
}

static P4TERM make_rational_rel(List *lst)
{ // make (& return) a conjunct of real/1 & rational/1 terms (e.g. real(X), rational(X)) for the
given list
P4TERM t, term;

for(term= NULL; lst; lst= lst->next)
        {
        t= P4MAKE_FUNC_1("rational", (P4TERM) lst->elem);
 term= P4AND(term, t);
        }
```

```
// add the actual real_rel-term to the VAR_TYPES buffer -  so we can later manipulate it
if (term)
        ADD_VAR_TYPES_TERM(term);

//      return just a placeholder since we already added the real_rel-term to the buffer
return(P4TRUE);
}

static P4TERM make_neqvars_rel(List *varslist)
{       // make a term to enforce all var's in the list are different from each other;
        //      put that term in VARS_TYPE_BUF for later use; return P4TRUE for now at end
P4TERM t, term;
List *list1, *list2;

term= NULL;
for(list1= varslist; list1; list1= list1->next)
        for(list2= list1->next; list2; list2= list2->next)
                {
                if (list1->elem && list2->elem)
                        {
                        t= P4NEQ((P4TERM)list1->elem, (P4TERM)list2->elem);
                        term= P4AND(term, t);
                        }
                }

// add the actual neqvars-term to the VAR_TYPES buffer -  so we can later manipulate it
if (term)
        ADD_VAR_TYPES_TERM(term);

//      return just a placeholder since we already added the neqvars-term to the buffer
return(P4TRUE);
}

static P4TERM make_eqvars_rel(List *varslist)
{       // make a term to enforce all var's in the list are equal to each other;
        //      put that term in VARS_TYPE_BUF for later use; return P4TRUE for now at end
P4TERM t, term;
List *list1, *list2;

term= NULL;
for(list1= varslist; list1; list1= list1->next)
        for(list2= list1->next; list2; list2= list2->next)
                {
                if (list1->elem && list2->elem)
                        {
```

```
                t= P4EQ((P4TERM)list1->elem, (P4TERM)list2->elem);
                term= P4AND(term, t);
                }
        }

// add the actual eqvars-term to the VAR_TYPES buffer -    so we can later manipulate it
if (term)
        ADD_VAR_TYPES_TERM(term);

//      return just a placeholder since we already added the eqvars-term to the buffer
return(P4TRUE);
}

static P4TERM make_neqvarvals_rel(P4TERM var, List *valslist)
{       // make a term to specify that the given var is not equal to any of the given values in the
valslist;
        //      put that term in VARS_TYPE_BUF for later use; return P4TRUE for now at end
P4TERM t, term;
List *list1;

if (!var || !valslist)
        return(P4TRUE);

term= NULL;
for(list1= valslist; list1; list1= list1->next)
        {
        t= P4NEQ(var, (P4TERM)list1->elem);
        term= P4AND(term, t);
        }

// add the actual eqvars-term to the VAR_TYPES buffer -    so we can later manipulate it
if (term)
        ADD_VAR_TYPES_TERM(term);

//      return just a placeholder since we already added the eqvars-term to the buffer
return(P4TRUE);
}

static P4TERM make_optimizable_rel(P4TERM rel)
{       // optimize the given rel-term (for now, we just put it at the start of the clauses);
        //      put that term in VARS_TYPE_BUF for later use; return P4TRUE for now at end
if (!rel)
        return(P4TRUE);

// add the actual term to the VAR_TYPES buffer - so we can later manipulate it
```

```
ADD_VAR_TYPES_TERM(rel);

//        return just a placeholder since we already added the term to the buffer
return(P4TRUE);
}

static P4TERM make_inlist(P4TERM var, List *lst)
{ // make (& return) (var inlist lst) constraint; (lst is list of P4 terms.);
        // (Obsolete: the translation uses inlist/1 e.g. "X in [a, b, 5]"  =>  "X ~ inlist([a, b, 5])".
        //                that approach provoked many problems in relations which expected constants e.g. gcd/2.)
        // Obsolete: return(P4MAKE_FUNC_2("inlist", var, make_termlist2term(lst)));
        // The current translation essentially uses disjunction e.g. "X in [a, b, 5]." => (X= a; X= b; X= 5)."

return(P4MAKE_FUNC_2("one_of", var, make_termlist2term(lst)));
}

static P4TERM make_fromlist(P4TERM var, List *lst)
{ // make (& return) (var inlist lst) constraint; (lst is list of P4 terms.);
        // some modifications/optimizations are made for randomization
        // mark the given var as an independent variable
TabElem *p_tab_elem;
P4TERM term;

if (!(p_tab_elem= get_term_tabelem(var)))
        return(NULL);
p_tab_elem->is_independent_var = TRUE;            // all enumerated var's are considered independent

term= P4MAKE_FUNC_2("random", var, make_termlist2term(lst));

// add the actual term to the enumRangeTerm buffer -
//                so we can later randomize it (to help produce different-looking solutions)
ADD_ENUM_RANGE_TERM(term);

//        return just a placeholder since we already added the enumRange term to the buffer
return(P4TRUE);

}

static P4TERM make_notinlist(P4TERM var, List *lst)
{ // make (& return) (var notinlist lst) constraint; (lst is list of P4 terms.);
        // ?? should we try the conjunction e.g. "X notin [a, b, 5]." => (X=/= a, X=/= b. X=/= 5)." ???
```

```
return(P4MAKE_FUNC_2("outlist", var, make_termlist2term(lst)));
}

static P4TERM make_interval(P4TERM left, int left_rel, P4TERM var, int right_rel, P4TERM
right)
{       // make (& return) interval constraint e.g. "4.5 < X <= 9"  => "oc(X, 4.5, 9)"; "4.6 >= X >
2" => "oc(X, 2, 4.6)"
        // Note that this is different from an enumeration over an interval.
char pred_name[3];
P4TERM term;

if (!var || ((left_rel==LE || left_rel==LT) && (right_rel==GE || right_rel==GT)) ||
                ((left_rel==GE || left_rel==GT) && (right_rel==LE || right_rel==LT)))
 {      // e.g. "4.5 < X > 7.5"
 semError= ERR_INVALID_INTERVAL;
 return(NULL);
 }
                // build the predicate name from the given relationships
pred_name[0]= (left_rel==LE || left_rel==GE)? 'c':'o';
pred_name[1]= (right_rel==LE || right_rel==GE)? 'c':'o';
pred_name[2]= 0;

term= P4MAKE_FUNC_3(pred_name, var, left, right);

#ifdef WANT_ALL_SMALL_SOLUTIONS
                // while this ok to do (and gives good solutions), it leads to unnecessary
undesirably small solutions in large quantity
        // --- For now, don't use it ---
if (useIntervalSolver)
        {               // split the bounded var when using interval-solver
        char *split_pred;
        P4TERM t, lst_term;
        TabElem *p_tab_elem;

        if (!(p_tab_elem= get_term_tabelem(var)))
                return(NULL);
        split_pred= (p_tab_elem->type== VAL_INTEGER)? "intsplit": "realsplit";
        lst_term= p4make_dot(var, p4make_nil());   // X -> [X]
        t= P4MAKE_FUNC_3(split_pred, lst_term,
p4make_atom_from_cstring("smallest_domain"), P4Make_Rational(p_tab_elem->precision));
        term= P4AND(term, t);
        }
#endif /* WANT_ALL_SMALL_SOLUTIONS */

return(term);
```

```
}

static P4TERM make_enumeration(P4TERM left, int left_rel, P4TERM var, int right_rel,
P4TERM right, P4TERM step)
{       // make (& return) enumeration constraint e.g. "[4.5 < X <= 8 step 1]"  => "(X=4.5;
X=5.5; X= 6.5; X= 7.5)"
        // Note that this is different from interval.
char *pred_name;
TabElem *p_tab_elem;
P4TERM term, adj_left, adj_right;

if (!var || !step || ((left_rel==LE || left_rel==LT) && (right_rel==GE || right_rel==GT)) ||
                    ((left_rel==GE || left_rel==GT) && (right_rel==LE || right_rel==LT)))
 {      // e.g. "4.5 < X > 7.5"
 semError= ERR_INVALID_INTERVAL;
 return(NULL);
 }

if (!(p_tab_elem= get_term_tabelem(var)))
        return(NULL);
p_tab_elem->is_independent_var = TRUE;          // all enumerated var's are considered
independent
// Note: we don't care here which is min or max value - enumerate/4 takes care of that.
pred_name= enumerateVarsRandomlyNoHistory? "enumerate_random": "enumerate";
if (p_tab_elem->type== VAL_INTEGER)
        pred_name= enumerateVarsRandomlyNoHistory?
"enumerate_int_random":"enumerate_int";

adj_left= (left_rel==LT)? P4MAKE_FUNC_2("+", left,
check_term(P4Make_Rational(p_tab_elem->precision))):
                                        (left_rel==GT)? P4MAKE_FUNC_2("-", left,
check_term(P4Make_Rational(p_tab_elem->precision))):
                                        left;
adj_right=(right_rel==LT)? P4MAKE_FUNC_2("+", right,
check_term(P4Make_Rational(p_tab_elem->precision))):
                                        (right_rel==GT)? P4MAKE_FUNC_2("-", right,
check_term(P4Make_Rational(p_tab_elem->precision))):
                                        right;

term= P4MAKE_FUNC_4(pred_name, var, adj_left, adj_right, step);
// add the actual enumerated-range term to the enumRangeTerm buffer -
//                  so we can later randomize it (to help produce different-looking solutions)
ADD_ENUM_RANGE_TERM(term);

//      return just a placeholder since we already added the enumRange term to the buffer
```

```
return(P4TRUE);
}

static P4TERM make_exterval(P4TERM left, int left_rel, P4TERM var, int right_rel, P4TERM right)
{ // make (& return) exterval constraint e.g. not("4.5 < X <= 9")  => "outoc(X, 4.5, 9)"; not("4.6 >= X > 2)" => "outoc(X, 2, 4.6)"
char pred_name[6];

if (!var || ((left_rel==LE || left_rel==LT) && (right_rel==GE || right_rel==GT)) ||
        ((left_rel==GE || left_rel==GT) && (right_rel==LE || right_rel==LT)))
  {     // e.g. "4.5 < X > 7.5"
  semError= ERR_INVALID_INTERVAL;
  return(NULL);
  }

strcpy(pred_name, "out");
pred_name[3]= (left_rel==LE || left_rel==GE)? 'c':'o';
pred_name[4]= (right_rel==LE || right_rel==GE)? 'c':'o';
pred_name[5]= 0;

return(P4MAKE_FUNC_3(pred_name, var, left, right));
}

static P4TERM make_functor(char *pred_name, List *arg_lst)
{       // make & return PrologIV functor for the given predicate/args (e.g. given "mean", [X, Y] for "mean(X, Y)")
        // We allow no user-relations - only user-functions.
        // As such, we add 1 result var (at the leftmost position) to all the user-functions.
        // free the given list after use ??
int i, cnt, arity;
P4TERM terms[MAX_ARITY+1], anon_result, func_term;
List *lst;

anon_result= NULL;
cnt= 0;
arity= arg_lst? arg_lst->elem_cnt: 0;
if (arity > MAX_ARITY)
  {
  semError= ERR_ARITY_TOO_MANY;
  return(NULL);
  }
         // convert given func/n to rel/n+1 e.g. X= mean(X, Y) -> X= _R, mean(_R, X, Y).
terms[cnt++]= anon_result= check_term(p4make_var());
```

```
for(i= 0, lst= arg_lst; i< MAX_ARITY && lst; lst= lst->next,i++)
  terms[cnt++]= (P4TERM)lst->elem;

pred_name= map_func_name(pred_name);
func_term= check_term(p4vmake_functor(cnt, p4str2symbol(pred_name), terms));
ADD_funcTermBuf(func_term);      // add func_term to funcTermBuf so we can generate code
at end for it
return(anon_result);
}


static P4TERM indexed_list_element(P4TERM list, List *index_lst)
{       // return PrologIV term for the given list-indexed-term (e.g. L[1, 2])
        // Translation scheme: List[I, J, K] => nth(K, nth(J, nth(I, List))).
P4TERM nth_elem, func_term;

if (!list || !index_lst)
        return(NULL);

nth_elem= check_term(p4make_var());
func_term= P4MAKE_FUNC_3("nth", nth_elem, index_lst->elem, list);
ADD_funcTermBuf(func_term);      // add func_term to funcTermBuf so we can generate code
at end for it

return(index_lst->next? indexed_list_element(nth_elem, index_lst->next): nth_elem);
}

static P4TERM if_then_else(P4TERM cond, P4TERM then_term, P4TERM else_term)
{ // make & return PrologIV term for the given if-then-else construct
 // This is a backtrack-less-implementation of if-then-else.
 //      if C then T else E => (C, T, !); E.

return(else_term? P4IF_THEN_ELSE(cond, then_term, else_term): P4IF_THEN(cond,
then_term));
}

static P4TERM if_then_elseif(P4TERM then_cond, P4TERM then_term, P4TERM else_cond,
P4TERM else_term)
{ // make & return PrologIV term for the guarded-if if-then-elseif construct
 //      if TC then T elseif EC E => (TC, T); (EC, E).

return(P4IF_THEN_ELSEIF(then_cond, then_term, else_cond, else_term));
}
```

```
%}

%union
{
 int ival;
 float fval;
 double dval;
 char *string;
 List *list;
 P4TERM term;
}

%token      <ival>      INTNUM NOTIN_SET IN_SET FROM_SET NOT PI
%token      <ival>      INT_PRED REAL_PRED RATIONAL_PRED
FRACTION_PRED LIST_PRED FREEZE
%token      <ival>      IF THEN ELSE ELSEIF SUCCEED FAIL SYMBOL_PRED
END_VAR_DEFS STEP
%token      <ival>      EQVARS_PRED NEQVARS_PRED NEQVARVALS_PRED
OPTIMIZABLEREL_PRED
%token      <ival>      ONGRID_PRED OFFGRID_PRED
%token      <string>    REALNUM ATOM_CONST VAR
%token      <ival>      GT LT
%token      <ival>      GE     ">="
%token      <ival>      LE     "<="
%token      <ival>      EQ "=="
%token      <ival>      NEQ "=/="
%token      <ival>      EXRANGE_START "[!"

%left       ';'
%left       ','
%nonassoc   '~'
%nonassoc   '='
%left       "==" '<' '>' ">=" "<="
%nonassoc   '|'
%left       '+' '-'
%left       '*'
%left       '/'
%left       '%' '\\'
%nonassoc   NEG
%left       '^'
%left       '!'

%type <term>  prolog_expr expr expr_lst rel_expr range_expr type_list type_symbol
%type <term>  rel_expr arith_expr function p_list type_int type_real type_rational type_fraction
%type <term>  constant num_constant atom var if_then_else flow_expr type_expr base_expr
```

```
%type <term> index type_eqvars type_neqvars type_neqvarvals type_optimizablerel
%type <term> type_ongrid type_offgrid
%type <ival>  rel_op_lege
%type <dval>  number
%type <list>  list var_lst constant_lst

%%
prolog_expr    :        expr_lst '.'                            {$$=
(P4TERM)process_call_to_prolog($1);}
                        |       expr_lst ',' END_VAR_DEFS ',' expr_lst '.'
          {$$= (P4TERM)process_call_to_prolog(insert_func_defs($1, $5));}
                        |       error
                        {$$= NULL; }
                        ;

expr_lst:       expr_lst ',' expr_lst               {$$= P4AND($1, $3);}
                        |       expr_lst ';' expr_lst                   {$$= P4OR($1, $3);}
                        |       expr
                        ;

expr    :'(' expr_lst ')'                                       {$$= $2;}
                        |       base_expr
                {$$= append_func_defs($1);}
                        ;

base_expr:      rel_expr
                        |       range_expr
                        |       flow_expr
                        |       type_expr
                        ;

flow_expr       :       if_then_else
                        |       FREEZE                                          {$$=
P4CUT;}
                        |       SUCCEED                                         {$$=
P4TRUE;}
                        |       FAIL                                            {$$=
P4FAIL;}
                        ;

type_expr       :       type_int
                        |       type_real
                        |       type_rational
                        |       type_fraction
                        |       type_list
```

```
                        |       type_symbol
                        |       type_ongrid
                        |       type_offgrid
                        |       type_eqvars
                        |       type_neqvars
                        |       type_neqvarvals
                        |       type_optimizablerel
                        ;

if_then_else:   IF expr THEN expr ELSE expr        {$$= if_then_else($2, $4, $6);}
                        |       IF expr THEN expr ELSEIF expr THEN expr         {$$=
if_then_elseif($2, $4, $6, $8);}
                        |       IF expr THEN expr
                        {$$= if_then_else($2, $4, NULL);}
                        ;

type_int        :       INT_PRED var_lst ')'        {mark_term_list_type($2,
VAL_INTEGER); $$= make_int_rel($2);}
                        ;
type_real       :       REAL_PRED var_lst ')'               {mark_term_list_type($2,
VAL_RATIONAL_FLOAT); $$= make_real_rel($2);}
                        ;
type_rational   :       RATIONAL_PRED var_lst ')'           {mark_term_list_type($2,
VAL_RATIONAL_FLOAT); $$= make_rational_rel($2);}
                        ;
type_fraction:  FRACTION_PRED var_lst ')'        {mark_term_list_type($2,
VAL_RATIONAL_FRACTION); $$= make_real_rel($2);}
                        ;
type_list       :       LIST_PRED var_lst ')'                           {$$= P4TRUE;
mark_term_list_type($2, VAL_LIST);}
                        ;
type_symbol     :       SYMBOL_PRED var_lst ')'             {$$= P4TRUE;
mark_term_list_type($2, VAL_SYMBOL);}
                        ;
type_ongrid:    ONGRID_PRED var ')'                         {$$= P4TRUE;
set_term_on_grid($2);}
                        ;
type_offgrid:   OFFGRID_PRED var ')'                {$$= P4TRUE;
reset_term_on_grid($2);}
                        ;

type_eqvars     :       EQVARS_PRED var_lst ')'                 {$$= make_eqvars_rel($2);}
                        ;
type_neqvars:   NEQVARS_PRED var_lst ')'        {$$= make_neqvars_rel($2);}
                        ;
```

```
type_neqvarvals:      NEQVARVALS_PRED var ',' list ')'          {$$=
make_neqvarvals_rel($2, $4);}
                      ;
type_optimizablerel:  OPTIMIZABLEREL_PRED rel_expr ')'          {$$=
make_optimizable_rel($2);}
                      ;

rel_expr        :     arith_expr '<' arith_expr       {$$= binop_bool_term($1, LT, $3);}
                      |     arith_expr '>' arith_expr                {$$=
binop_bool_term($1, GT, $3);}
                      |     arith_expr "==" arith_expr      {$$= binop_bool_term($1,
EQ, $3);}
                      |     arith_expr "=/=" arith_expr     {$$= binop_bool_term($1,
NEQ, $3);}
                      |     arith_expr '=' arith_expr                {$$=
binop_bool_term($1, EQ, $3);}
                      |     arith_expr ">=" arith_expr      {$$= binop_bool_term($1,
GE, $3);}
                      |     arith_expr "<=" arith_expr      {$$= binop_bool_term($1,
LE, $3);}
                      |     '(' rel_expr ')'                              {$$=
$2;}
                      | NOT rel_expr
              {$$= P4NOT($2);}
/*                    |     function                          {$$= $1;} --
may not have relations */
                      ;

range_expr      :     '[' arith_expr rel_op_lege var rel_op_lege arith_expr ']'

                                  {$$= make_interval($2, $3, $4, $5, $6);}
                      |     "[!" arith_expr rel_op_lege var rel_op_lege arith_expr ']'

                                  {$$= make_exterval($2, $3, $4, $5, $6);}
                      |     '[' arith_expr rel_op_lege var rel_op_lege arith_expr STEP
arith_expr ']'

                                  {$$= make_enumeration($2, $3, $4, $5, $6, $8);}
                      |     var FROM_SET '[' list ']'         {$$= make_fromlist($1,
$4);}
                      |     var IN_SET '[' list ']'           {$$= make_inlist($1, $4);}
                      |     var NOTIN_SET '[' list ']'    {$$= make_notinlist($1, $4);}
                      ;

rel_op_lege     :     '<'
```

```
        {$$= LT;}
                        |       "<="
                                {$$= LE;}
                        |       '>'
                                        {$$= GT;}
                        |       ">="
                                {$$= GE;}
                        ;

p_list:  '[' list ']'                                           {$$=
make_termlist2term($2);}
                        ;

list    :       list ',' arith_expr                     {$$= ncons($1, (void *)$3);}
                        |       arith_expr
                {$$= list((void *)$1);}
                        ;

arith_expr      :       arith_expr '+' arith_expr       {$$= binop_arith_term($1, '+', $3);}
                        |       arith_expr '-' arith_expr                       {$$=
binop_arith_term($1, '-', $3);}
                        |       arith_expr '/' arith_expr                       {$$=
binop_arith_term($1, '/', $3);}
                        |       arith_expr '*' arith_expr                       {$$=
binop_arith_term($1, '*', $3);}
                        |       arith_expr '^' arith_expr                       {$$=
binop_arith_term($1, '^', $3);}
                        |       arith_expr '%' arith_expr                       {$$=
make_functor("mod", cons((void *)$1, list((void *)$3)));}
                        |       arith_expr '\\' arith_expr              {$$=
make_functor("intdiv", cons((void *)$1, list((void *)$3)));}
                        |       '-' arith_expr     %prec NEG            {$$= unop_term('-',
$2);}
                        |       arith_expr '!'
                {$$= make_functor("factorial", list((void *)$1));}
                        |       '|' arith_expr '|'                                      {$$=
make_functor("abs", list((void *)$2));}
                        |       '(' arith_expr ')'
        {$$= $2;}
                        |       function
                                        {$$= $1;}
                        |       index
                                                {$$= $1;}
                        |       p_list
                                        {$$= $1;}
```

```
                        |       constant
                                        {$$= $1;}
                        |       var
                                                        {$$= $1;}
                        ;

function        :       ATOM_CONST '(' list ')'                 {$$= make_functor($1, $3);}
                        |       ATOM_CONST '(' ')'
        {$$= make_functor($1, NULL);}
                        ;

index   :       var '[' list ']'                                        {$$=
indexed_list_element($1, $3);}
                        ;

var_lst         :       var_lst ',' var                                 {$$=
ncons($1, (void *)$3);}
                        |       var
                                                        {$$= list((void *)$1);}
                        ;

constant_lst:   constant_lst ',' constant               {$$= ncons($1, (void *)$3);}
                        |       constant
                                                {$$= list((void *)$1);}
                        ;

constant        :       num_constant
                        |       atom
                        ;

num_constant:           INTNUM                          {Value val; $$=
check_term(p4make_lint($1)); val.type= VAL_INTEGER; val.value.integer= $1;
insert_const($$, &val);}
                        |       REALNUM                                 {Value val;
$$= check_term(p4make_rational($1)); /* yylex passes string in yylval */ val.type=
VAL_RATIONAL_FLOAT; val.value.rational.real= atof($1); insert_const($$, &val);}
                        |       PI                                              {Value
val; $$= check_term(P4Make_Rational(PI_VAL)); val.type= VAL_RATIONAL_FLOAT;
val.value.rational.real= PI_VAL; insert_const($$, &val);}
                        ;

atom    :       ATOM_CONST                              {$$=
check_term(p4make_atom_from_cstring($1));}
                        ;
```

```
var             :       VAR
                                                {$$= insert_var($1);}
                        |       '{' VAR ',' number '}'                          {$$=
insert_var_with_precision($2, $4);}
                        |       '{' VAR ',' number '/' number '}'       {$$=
insert_var_with_precision($2, ((double)$4)/$6);}
                        ;

number:         INTNUM                                  {$$= (double)$1;}
                        |       REALNUM                                 {$$=
(double)atof($1); /* yylex passes string in yylval */}
                        |       PI                                              {$$=
(double)PI_VAL;}
                        ;


%%
static int restart_parser()
{
yyclearin;
return(1);
}

static int restart_lexer()
{
extern void yyrestart(FILE *);

yyrestart(NULL);
return(1);
}

static BOOLEAN init_solve_constraint(int keep_solns)
{ // initialize all the buffers; if keep_solns is true, do not initialize the solns (& value) buffers
        // returns true if ok, false in error
if (!StartProlog4Session(NULL))
        return(FALSE);
 // initialize various data structures
if (!keep_solns)
        {
        INIT_valBuf;
        INIT_solnBuf;
        }
INIT_inExprBuf;
INIT_vars;
INIT_funcTermBuf;
```

```
INIT_anonVarBuf;
INIT_constBuf;
INIT_enumRangeTermBuf;
INIT_varTypesTermBuf;

 // initilize parser/lexer states & data structures (if any) ...
restart_lexer();
restart_parser();

return(TRUE);
}

int yyerror(char *s)
{ // handle error
// printf("syntax error\n");
semError= ERR_PARSE;
return(0);
}

        // some augmented functions for Prolog IV API - some of them may go away as we get
newer, better API
static P4SYMBOL p4str2symbol(char *str)     // pseudo p4-routine
{ // convert the given atomic (i.e. starting with lowercase e.g. 'aTom') string to Prolog IV symbol
P4SYMBOL sym;
if (p4cstring_to_symbol(str, &sym))
  {
  semError= ERR_GETTING_TERM;
  return(0);
  }
return(sym);
}

static P4TERM p4make_atom_from_cstring(char *str)           // pseudo p4-routine
{ // convert the given atomic (i.e. starting with lowercase e.g. 'aTom') C-string to Prolog IV
atomic term
P4SYMBOL sym;
if (p4cstring_to_symbol(str, &sym))
  {
  semError= ERR_GETTING_TERM;
  return(NULL);
  }
return(p4make_atom(sym));
}

static int p4is_constant(P4TERM term)            // pseudo p4-routine
```

```
{ // returns TRUE if the given term is constant
int type;

type= P4WHAT_IS(term);
return(!(type==P4NULL || type==P4NOTATERM || type== P4VAR || type==
P4UNEXPECTEDTERM));
}

static P4TERM P4Make_Rational(double val)                         // pseudo p4-routine
{ // KNJ: make (& return) term for the given double (rational) value
char buf[256];

sprintf(buf, "%.6f", val);        // maximum precision is limited to 6 positions
return(make_rational_strfloat(buf));
}

static P4TERM p4make_rational(char *floatstring)
{       // by Pascal Bouvier (of Prologianet): Build a positive rational number from a C-string
containing its decimal
        // representation (e.g. "4.5", "1.000000000", "24.02e1997", ...)
char buf[256];

strcpy(buf, floatstring);
return(make_rational_strfloat(buf));
}

static double p4val_as_double(P4TERM T)
{       // by Pascal Bouvier (of Prologianet):Given a numeric Term, converts it as a double (with
probable loss of precision)
switch (P4WHAT_IS(T))
  {
  case P4INTEGER: case P4RATIONAL:
              return(nearest_ip_fpd(dereference(T)));
  default:
              return(-11111.111); /* in error */
  }
}

static char *p4_symbol_to_cstring(P4SYMBOL symbol)
{       // by Pascal Bouvier (of Prologianet): return the c-string rep. of the symbol
return(symbol_shortidP(symbol));
}
```

```
' hlP4API.h
/*
 * hlP4API.h: Specification of high level API to Prolog IV
 *
 */

#include <windows.h>                          // necessary for VB stuff e.g. BSTR

// The stdcall calling-convention is used for compatibility with VB
#define CCONV _stdcall

#define DEF_PRECISION                                   (0.01)
#define DEF_SOLN_DIFF_WT                        (1)
#define DEF_FLOAT_INTERVAL_STEP                 (0.1)
#define DEF_INTEGER_INTERVAL_STEP               (1)
#define DEF_UPPER_BOUND                         (64000)
#define DEF_LOWER_BOUND                         (-64000)
#define DEF_GRID                        (TRUE)

typedef struct s_list
{  // ordered list
 short elem_cnt;        // total no. of elements (including this element) in the list
 void *elem;            // this element (type to be inferred from the context)
 struct s_list *next;
} List;

typedef struct s_functor
{       // structure to represent a Prolog IV functor (e.g. member/2) in C
 char *predicate;
 int arity;
} Functor;

typedef struct s_rational
{       // structure to represent a rational number
double real;    // real representation of a rational e.g. "4.5"
long num;               // numerator from A/B rep. of rational e.g. 9 from 9/2
long den;               // denominator from A/B rep. of rational e.g. 2 from 9/2
} Rational;

typedef struct s_bound          // a bound for non-rational real
{
 char is_infinite;              // flag - true if the real-val is infinite
 double val;                    // value of the non-infinite real
} Bound;
```

```
typedef struct s_real           // representation (bound) for non-rational real
{
 Bound lower, upper;
} Real;

typedef struct s_val
{
 int type;                      // type of the result (e.g. VAL_INTEGER, VAL_LIST)
 union {
 long integer;        // e.g. 5
 Rational rational;   // e.g. 9/2 = 4.5
 Real real;                     // e.g. (lower: 2.5, upper: 5)
 char *string;        // atom e.g. area
 Functor functor;               // e.g. add(X, Y)
 List *list;                    // value itself is a list of values e.g. [a, [x, y], 5]
 } value;
} Value;

        // Types of value
#define VAL_UNKNOWN             0
#define VAL_INTEGER             10
#define VAL_RATIONAL_FLOAT              12
#define VAL_RATIONAL_FRACTION           13
#define VAL_IRRATIONAL                  14
#define VAL_REAL        15              // not used in the Value structure
#define VAL_STRING              20
#define VAL_LIST        25
#define VAL_FUNCTOR             30
#define VAL_SYMBOL              35
#define VAL_VAR                 100
#define DEF_VAR_TYPE    VAL_UNKNOWN     // default type for untyped variables

char * CCONV GetHLAPIVersion(); // return the current version of the Prolog HL API
BSTR CCONV VBGetHLAPIVersion(); // VB wrapper for GetHLAPIVersion()

// StartProlog4Session: starts Prolog IV, return true (1) if ok, false (0) otherwise.
        // p4hlapilib_file is the pathname to the high-level Prolog IV API library file
        //      (MUST call one of: {StartProlog4Session, StartProlog4SessionSetStacks} before
starting Prolog IV.)
int CCONV StartProlog4Session(char *p4hlapilib_file);
// StartProlog4SessionSetStacks: starts Prolog IV, return true (1) if ok, false (0) otherwise.
        // p4hlapilib_file is the pathname to the high-level Prolog IV API library file
        //      heapsize is the heap-stack size; choicesize is the choice-stack size.
        //      (MUST call one of: {StartProlog4Session, StartProlog4SessionSetStacks} before
starting Prolog IV.)
```

```
int CCONV StartProlog4SessionSetStacks(char *p4hlapilib_file, long heapsize, long choicesize);

        // Error return-values from SolveConstraint() (keep them all negative)
#define ERR_INITIALIZATION                -10
#define ERR_CONSTRAINT_TOO_LONG                -15
#define ERR_GETTING_TERM                -20
#define ERR_MAKING_FUNCTOR                -25
#define ERR_INVALID_INTERVAL                -30
#define ERR_ARITY_TOO_MANY                -35
#define ERR_PARSE                -40
#define ERR_NULL_TERM                -45

// SolveConstraint: solve the given constraint (e.g. "X= Y+ 4, Y=2.") using a linear/interval
solver as needed;
        // backtrack over the previous solution if the given constraint is NULL.
        // return true (=1) [false (=0)] if the constraint is [un]solvable;
        //        returns negative integer in error (e.g. if the constraint could not be parsed).
int CCONV SolveConstraint(char *constraint);

// SolveConstraintLin: solve the given constraint (e.g. "X= Y+ 4, Y=2.") using a linear solver
only;
        // backtrack over the previous solution if the given constraint is NULL.
        // return true (=1) [false (=0)] if the constraint is [un]solvable;
        //        returns negative integer in error (e.g. if the constraint could not be parsed).
int CCONV SolveConstraintLin(char *constraint);

// SolveConstraintOrdered: solve the given constraint (e.g. "X= Y+ 4, Y=2."); using a
linear/interval solver as needed
        // present the solutions conforming to the given order (e.g. ORDER_DIFF_TOGETHER)
        // backtrack over the previous solution if the given constraint is NULL.
        // returns, on first call (i.e. when constraint is non-NULL), (1+
the_total_count_of_solutions) (> 0) [false (=0)] if the constraint is [un]solvable;
        //                (note that in case of constraints without variables (e.g. "4= 4."),
total-no.-of-solutions is 0, though the constraint is provable.)
        // returns, on subsequent calls (i.e. when constraint is NULL), true (= 1) [false (=0)] if a
solution exists [does not exist];
        //        returns negative integer in error (e.g. if the constraint could not be parsed).
int CCONV SolveConstraintOrdered(char *constraint, int order_type);

// SolveConstraintOrderedNSolns: solve the given constraint (e.g. "X= Y+ 4, Y=2."); using a
linear/interval solver as needed
        // solve to find the given maximum no. (= max_soln) of solutions
        // backtrack over the previous solution if the given constraint is NULL.
        // present the solutions conforming to the given order (e.g. ORDER_DIFF_TOGETHER)
        // returns, on first call (i.e. when constraint is non-NULL), (1+
```

the_total_count_of_solutions) (> 0) [false (=0)] if the constraint is [un]solvable;
// (note that in case of constraints without variables (e.g. "4= 4."), total-no.-of-solutions is 0, though the constraint is provable.)
// returns, on subsequent calls (i.e. when constraint is NULL), true (= 1) [false (=0)] if a solution exists [does not exist];
// returns negative integer in error (e.g. if the constraint could not be parsed).
int CCONV SolveConstraintOrderedNSolns(char *constraint, int order_type, int max_soln);

// order-types
#define NO_ORDER 0
#define ORDER_DIFF_TOGETHER 10
#define ORDER_LIKE_TOGETHER 20
#define ORDER_RANDOM 30
#define ORDER_UNIQ_SOLUTIONS 40

// set the precision for solving the constraint & for the solutions in the real domain
// returns TRUE if ok, FALSE otherwise
int CCONV SetPrecision(double precision);

// set the weight to indicate how "different" the solutions must be from each other in Uniq_Soln_Order
// (the higher the weight, the more the solutions are "different".)
// returns TRUE if ok
int CCONV SetSolnDiffWt(int soln_diff_wt);

// fractionalize the rationals if do_fractionalize is TRUE; not otherwise (fractionalization may slow things down a bit.)
int CCONV FractionalizeRational(int do_fractionalize);

// IsFullyConstrained: returns TRUE if the given constraint is fully constrained (i.e. solvable & all variables are constant); FALSE otherwise (or in error)
int CCONV IsFullyConstrained(char *constraint);

// return TRUE (1) if the given variable is independent (i.e. specified in an enumeration); FALSE (0) otherwise
int CCONV IsIndependentVar(char *var);

// GetValue: return the ptr to the value (in Value structure) of the given variable (e.g. "Area") if known;
Value * CCONV GetValue(char *var);

// return type (e.g. VAL_INTEGER) of the given Value;
// returns VAL_UNKNOWN in error
long CCONV GetValue_type(Value *val);

```
        // return type (e.g. VAL_INTEGER) of the given variable;
                //      returns VAL_UNKNOWN in error
long CCONV GetVarValue_type(char *var);

// GetValue_int: return integer value of the given Value structure
        // return ERR_GETVALUE_INT in error (e.g. given structure is not integer)
long CCONV GetValue_int(Value *value);
#define ERR_GETVALUE_INT        (-111111765)

// GetValue_rational: return rational value of the given Value structure
        // return <ERR_GETVALUE_RAT, ERR_GETVALUE_INT, ERR_GETVALUE_INT>
in error (e.g. given structure is not rational)
Rational CCONV GetValue_rational(Value *value);
#define ERR_GETVALUE_RAT        (-111111765.9876)

        // return float rep. of the given rational Value
double CCONV GetValue_rational_float(Value *val);

        // return numerator of the fractional rep. of the given rational Value
long CCONV GetValue_rational_numer(Value *val);

        // return denominator of the fractional rep. of the given rational Value
long CCONV GetValue_rational_denom(Value *val);

        // return real value (i.e. lower & upper bound) from the given non-rational Value
structure;
                        //      return <1, 0> in error (e.g. given structure is not real)
Real CCONV GetValue_real(Value *val);

#define ERR_GETVALUE_REAL       (-111111765.9876)

        // return lower bound for the given non-rational real value
                // return ERR_GETVALUE_REAL in error or when the lower bound is infinite
double CCONV GetValue_real_lower(Value *val);

        // return upper bound for the given non-rational real value
                // return ERR_GETVALUE_REAL in error or when the upper bound is infinite
double CCONV GetValue_real_upper(Value *val);

// GetValue_string: return (uniform) string representation of the given Value structure
        // return NULL in error (e.g. given structure is not valid)
char * CCONV GetValue_string(Value *value);
BSTR CCONV VBGetValue_string(Value *value); // VB wrapper for GetValue_string()

// GetVarValue: return (uniform) string representation of the given variable
```

```
        // return NULL in error
char * CCONV GetVarValue(char *var);
BSTR CCONV VBGetVarValue(char *var);          // VB wrapper for GetVarValue()
                    // return (uniform) string representation of the given variable in the value-buffer
int CCONV GetVarValueBuf(char *var, int valuebuf_len, char valuebuf[]);          // return length (>0) of value-string if ok; <= 0 in error

// PrintAllVarVals: Print all the var's with their values in the given buffer; return ptr to the given buffer
 //  (assumes the buffer is large enough to store all the var-value pairs.)
char * CCONV PrintAllVarVals(char buf[]);
BSTR CCONV VBPrintAllVarVals();          // VB wrapper (almost) for PrintAllVarVals()

// PrintAllVarVals: Print all the var's with their values in an allocated buffer; return ptr to the given buffer
 //  (assumes the buffer is large enough to store all the var-value pairs.)
char * CCONV PrintAllVarValsAllocate();

// Compile:compile & load the given p4_filename (containing Prolog IV program)
  // return true (1) if the file compiled ok, false (0) in error
int CCONV Compile(char *p4_filename);
```